Operating Systems (2INC0)

File System

Dr. Mitra Nasri

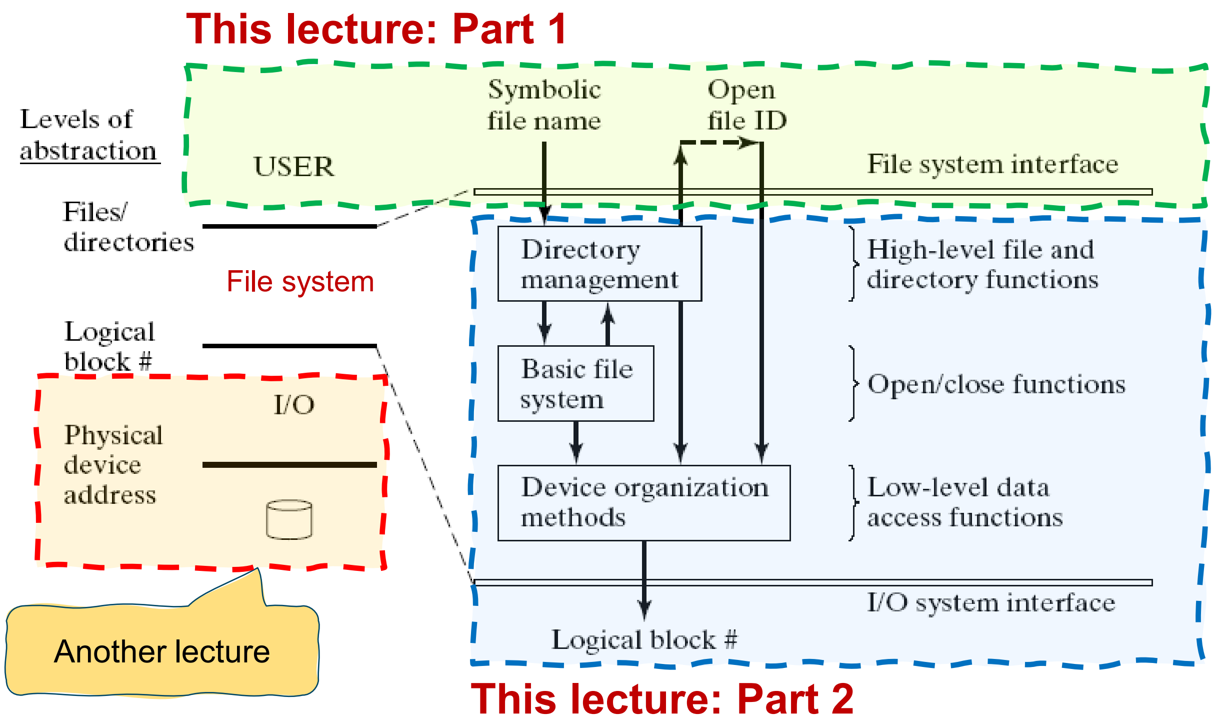
Interconnected
Resource-aware
Intelligent Systems

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

Agenda

- **Objectives of a File System**
- **Part 1: User view (file-system interface)**
- **Part 2: OS view (implementation of a file system)**



Sources

- Some slides have been made with the help of material designed by
  - Dr. Tanir Ozcelebi (from IRIS)
  - Dr. John Bell (University of Illinois)

- Helpful notes:
  - https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/11_FileSystemInterface.html
  - https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/12_FileSystemImplementation.html

- Textbook:
  - Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 11 and 12

Responsibilities of a file system

What file systems do you know?
What do they do in an OS?

FAT (FAT16, FAT32), NTFS, EXT4, UFS, …

The file system presents a **logical**, **abstract**, **coherent** view of **files** & **directories** and has three properties:

1. *persistent* across shutdowns
2. *independent* of (virtual) memory size
3. *facilitates sharing* between users, processes (not necessarily concurrent)

Basic requirements of a file system

- *naming*
- *transparency + portability* (hide hardware details through abstraction)
- *robustness* (against faults)
- *access control* (authorization, authentication)
- *security* (protection of user's files, data integrity)

User requirements of a file system

- Access files using a symbolic name
- Capability to create, delete, read, and change files

- Controlled access to system and other users’ files
- Control own access rights

- Capability of restructuring files (moving, copying, …)
- Capability to move data between files

- Backup and recovery of files

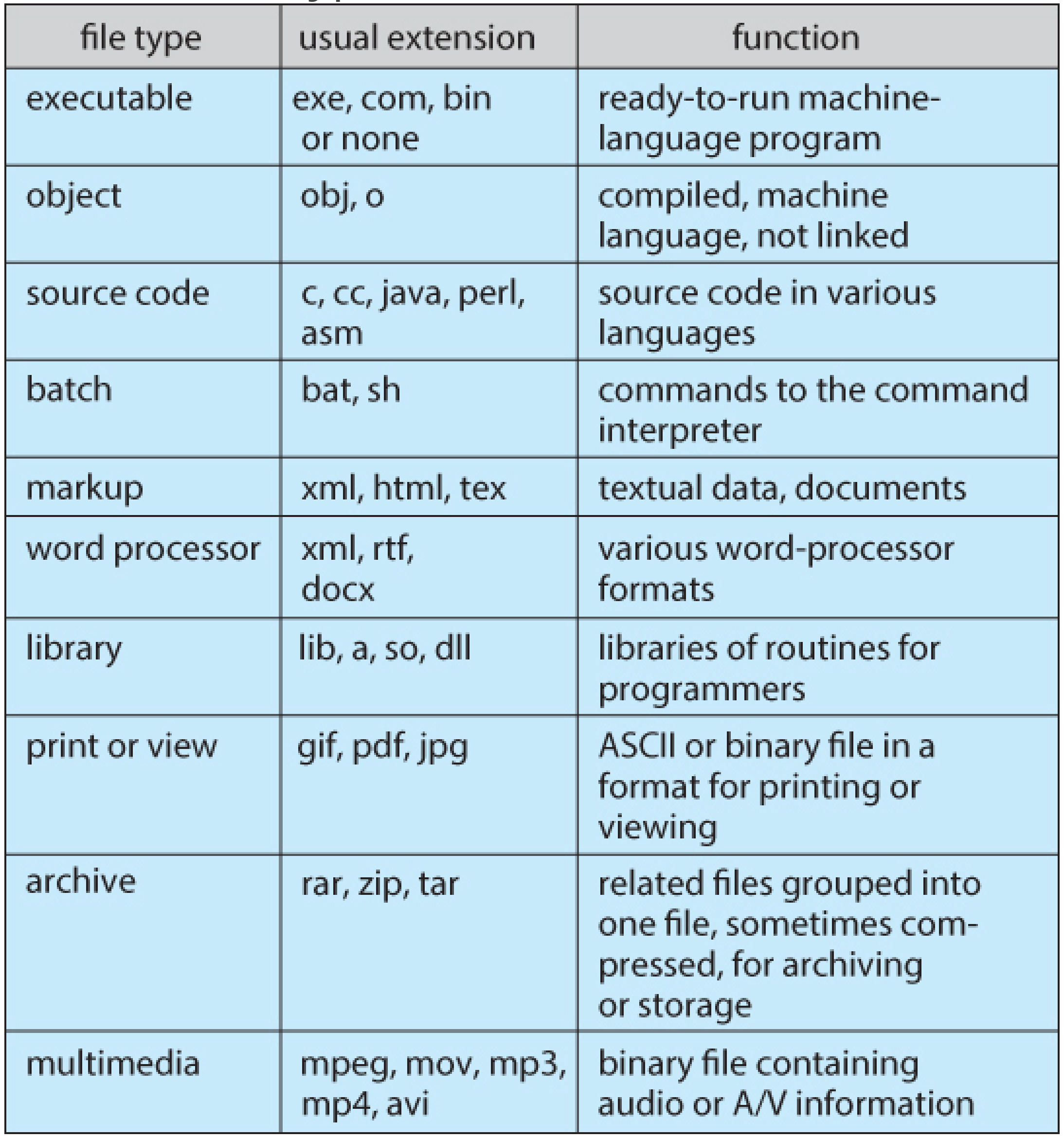
Files

File is a **logical data storage unit**, that provides an abstraction from physical properties of storage devices.

- Types:
  - numeric data
  - character data
  - binary data
  - program file

OS may or may not be aware of the internal structure of the file (it knows zip, txt, exe, but does not know dat)

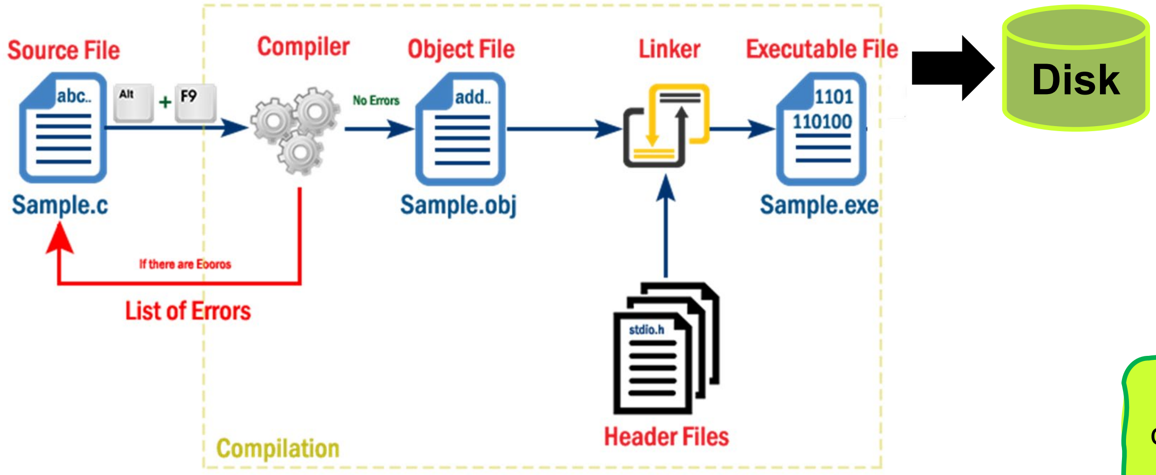
Common file types



file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

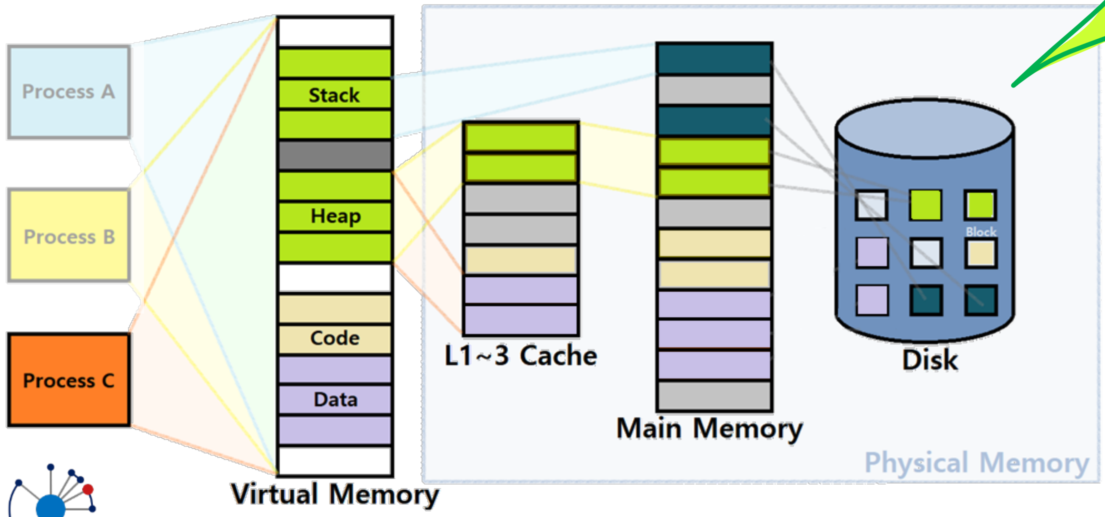
Recap: executing programs and the role of the secondary storage (disk)

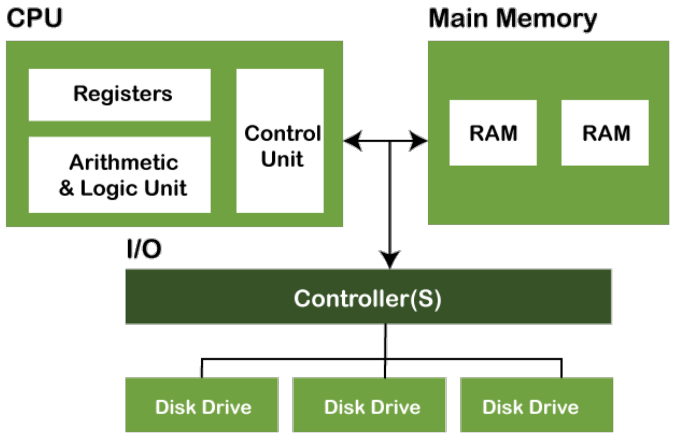
Creating a program:



And don't forget that a program can open/create other files and manipulate them during its execution.

During the execution of a process:





Views of file systems

- *User view* concerns
  - what constitutes a file
  - naming of files
  - allowed operations on files
  - …
- *OS view* (implementation) concerns
  - how should logical blocks of the disk be assigned to files?
  - how to keep track of free storage?
  - how to assign new blocks from the disk to a growing file?
  - …

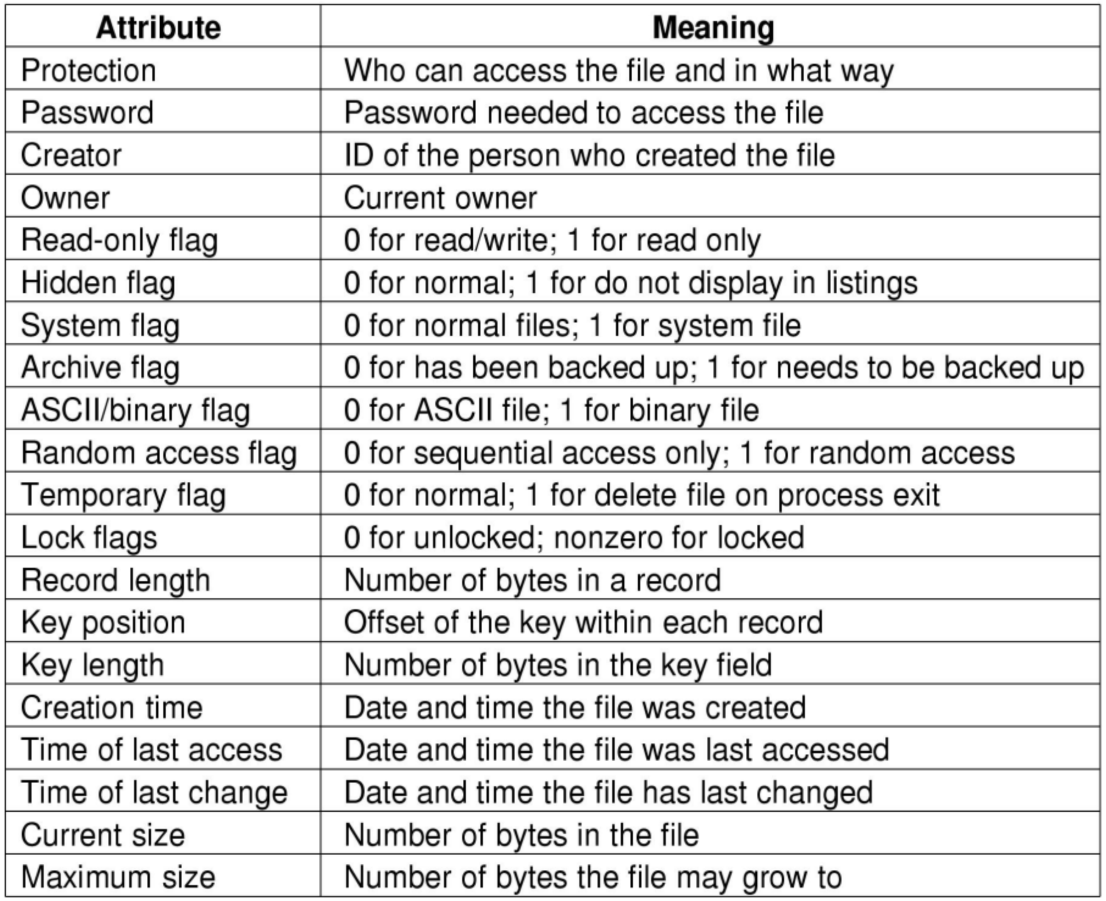
Agenda

- Objectives of a file system
- User view (file system interface)
  - File attributes
  - File access methods
  - Organization of directories
  - Access rights and access control
- OS view (implementation of a file system)

File attributes

- **Permanent attributes**, created upon file creation (attributes are permanent, not their values).
  - type
  - ownership
  - size
  - permanent/temporary flag
  - protection, access rights
  - dates
    - creation, access, modification

- **Temporary attributes**, created and maintained upon access.
  - read pointer, write pointer, buffers
  - open count
  - locking



Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

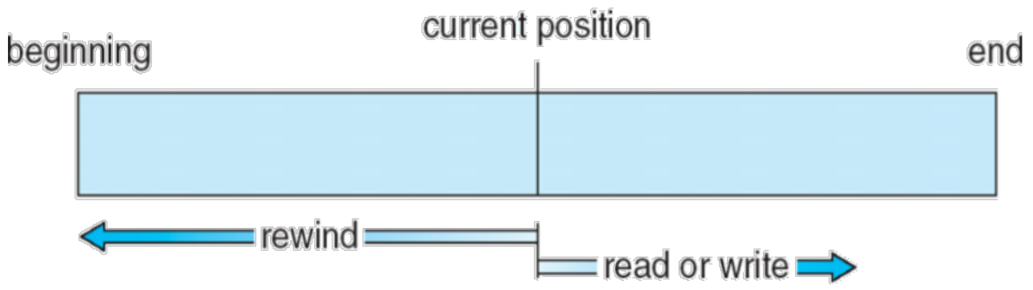
File access methods: Sequential access

File access should be compliant with the hardware.

1. Sequential access

- read bytes/records from the beginning
- cannot jump around (can only rewind)
- convenient when medium was "tape"

Sequential-access

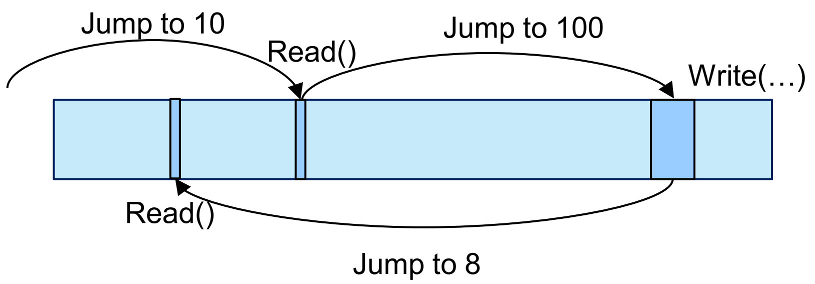


Permitted operations:

- **read next** - read a record and advance the tape to the next position.
- **write next** - write a record and advance the tape to the next position.
- **rewind**
- **skip n records** - May or may not be supported.
  - n may be limited to positive numbers or may be limited to +/- 1.

File access methods: Direct access

File access should be compliant with the hardware.



2. Direct access (or random access)

Can jump to any record and read that record or write on it.

- move current position (seek), then read
- or given a start position, read read read…

Permitted operations

- **read n** - read record number n (it requires an argument)
- **write n** - write record number n (it requires an argument)
- **jump to record n** - could be 0 or the end of file.

Note: Sequential access can be emulated using direct access.
The inverse is complicated and inefficient. (cp = current pointer)

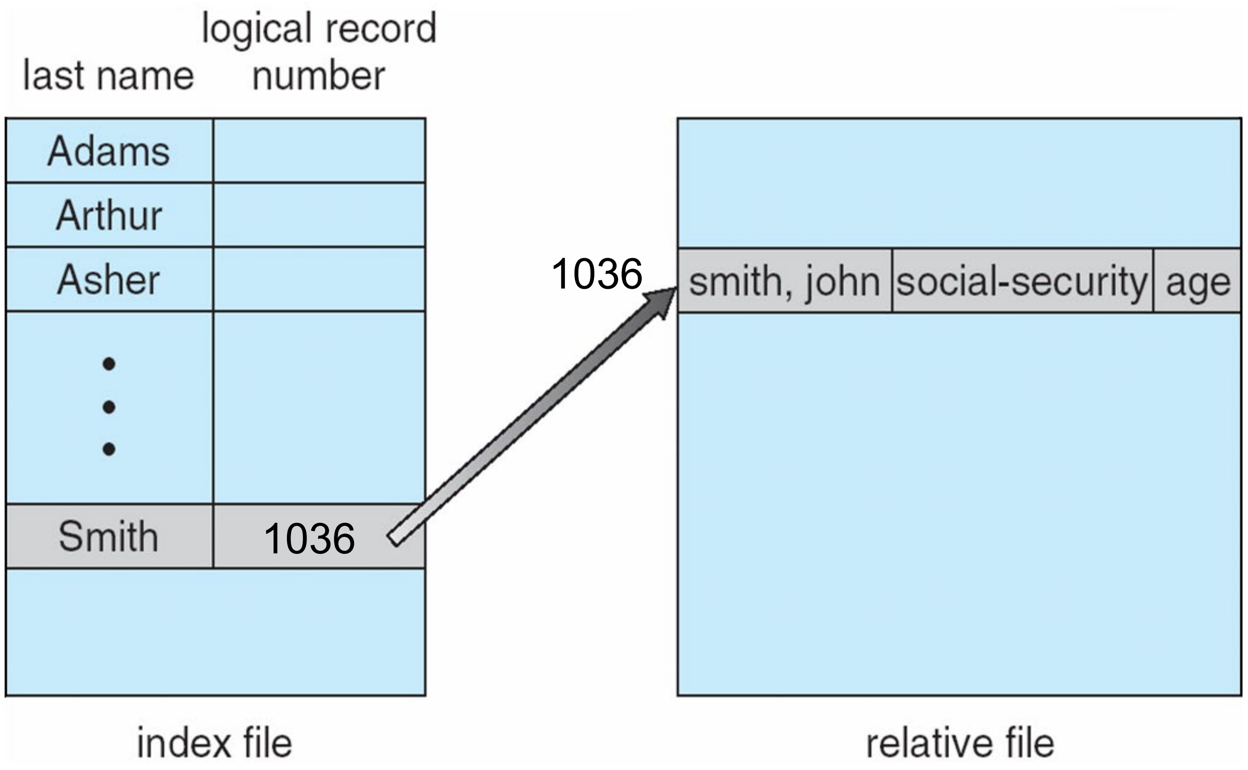
sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

File access methods: Through index files

File access should be compliant with the hardware.

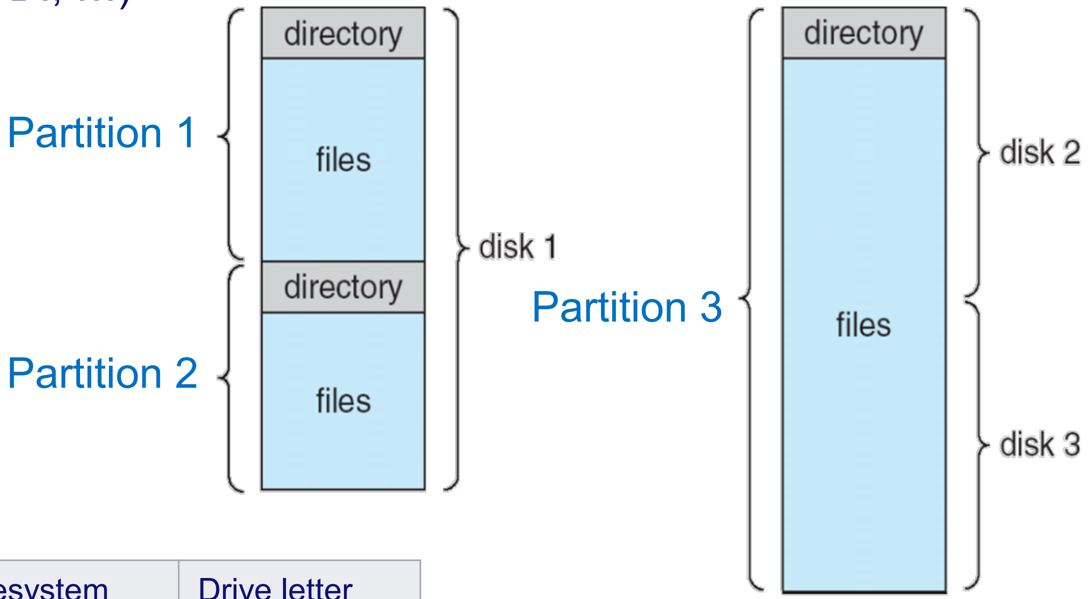
3. Access through index files

- built on top of direct-access method
- involves the construction of an index of the file
  - first search the index
  - then use the pointer to access the file directly



File system organization
Directory, volume, disk

- Directory: a *file* containing information about other files (and directories)
  - Both the directory structure and the files reside on disk.
- A disk can be subdivided into partitions (hardware portion)
- A partition can involve multiple disks
- A volume is a logical storage space (partition) that is formatted with a file system. It is referred to by a logical drive letter (e.g., C:, D:, …)
  - C:\... → MS-DOS volume



Each partition can have its own file system.

Physical disk	Partition	Filesystem	Drive letter
Hard Disk 1	Partition 1	NTFS	C:
	Partition 2	FAT32	D:
Hard Disk 2	Partition 3	FAT32	E:
Hard Disk 3			

In this example,
- "C:", "D:", and "E:" are volumes.
- Hard Disk 1, 2 and 3 are physical disks.
- Any of these can be called a "drive".

Directory operations semantics

- *Create:*
  - generate an empty structure (needs a ***parent directory*** to make it in)
    - typically, the *current working directory*
- *Delete:*
  - remove entry (a file or a subdirectory);
    - possibly: remove sub-tree, recursively
- *Rename:*
  - new name to a file or sub-directory
- *List:*
  - List the contents
- *Search for a file or sub-directory:*
  - recursive query through set of directories

Main API operations for regular files

Types of access to the file

- *Create / delete file*
  - insert/remove file into/from system
- *Open / close*
  - create and initialize (/destroy) *file descriptor*
    - descriptor: handle on the file: reference to attributes
- *Read / write*
  - access file at 'current position'
- *Seek*
  - change 'current position'
- Inspection and modification of *attributes*
  - e.g., change access rights, protection
- Explicit *synchronization* with memory
  - flush disk cache (embedded disk buffer, between disk and the main memory), or just the blocks of this file
    - Linux: *sync()* and *fsync()*

Access control

- File owner/creator should be able to control:
  - what can be done to the file
  - by whom
    - a user or a group of users, or
    - a process run by a user or a group of users

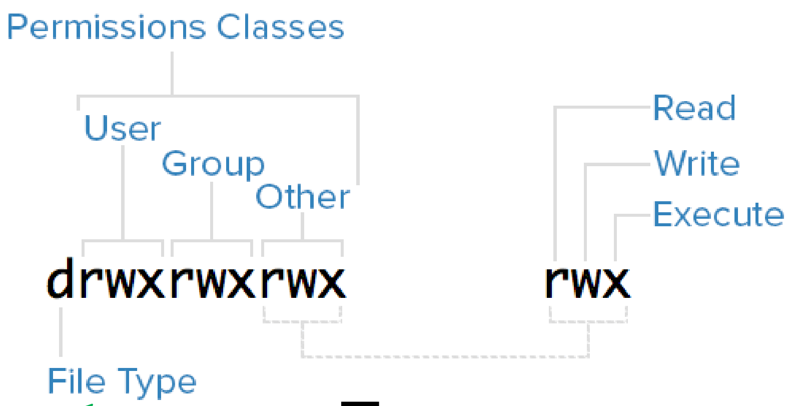
UNIX access lists and groups

- **Mode of access**:
  1. read (*R*)
  2. write (*W*)
  3. execute (*X*)

Three classes of user accesses:

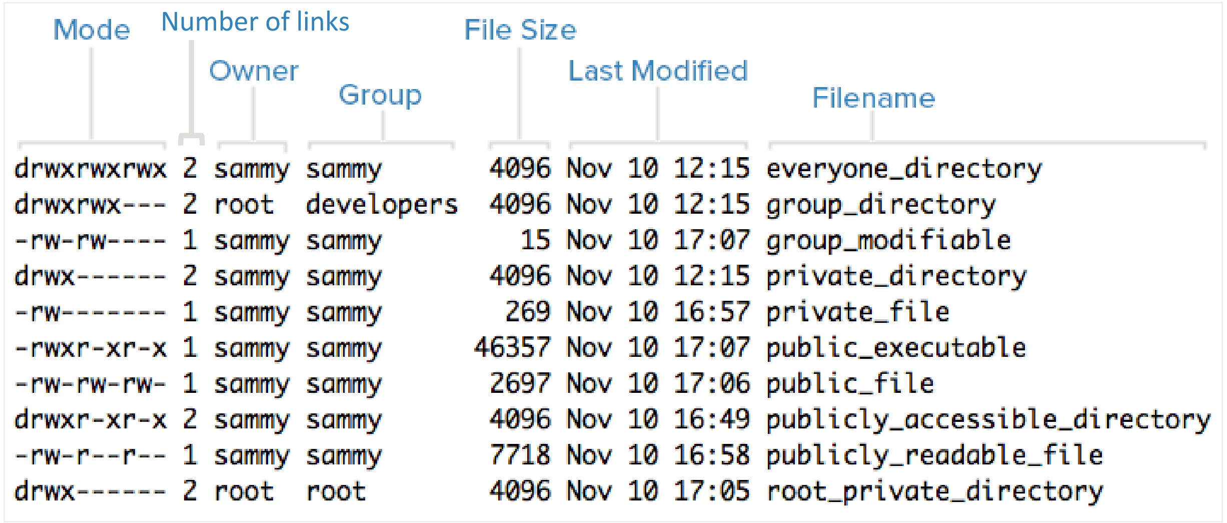
1. owner access (defines the rights of the owner)
2. group access (defines the rights of a group of users to which the file is assigned)
3. other (defines rights of any other user)

- Create a group (unique name), say developers, and add some users to the group.
- Attach group to a file (say "game") or subdirectory
  → chgrp developers game



- Normal files: -
- Directory: d

Example



```
drwxrwxrwx 2 sammy sammy       4096 Nov 10 12:15 everyone_directory
drwxrwx--- 2 root  developers  4096 Nov 10 12:15 group_directory
-rw-rw---- 1 sammy sammy         15 Nov 10 17:07 group_modifiable
drwx------ 2 sammy sammy       4096 Nov 10 12:15 private_directory
-rw------- 1 sammy sammy        269 Nov 10 16:57 private_file
-rwxr-xr-x 1 sammy sammy      46357 Nov 10 17:07 public_executable
-rw-rw-rw- 1 sammy sammy       2697 Nov 10 17:06 public_file
drwxr-xr-x 2 sammy sammy       4096 Nov 10 16:49 publicly_accessible_directory
-rw-r--r-- 1 sammy sammy       7718 Nov 10 16:58 publicly_readable_file
drwx------ 2 root  root        4096 Nov 10 17:05 root_private_directory
```

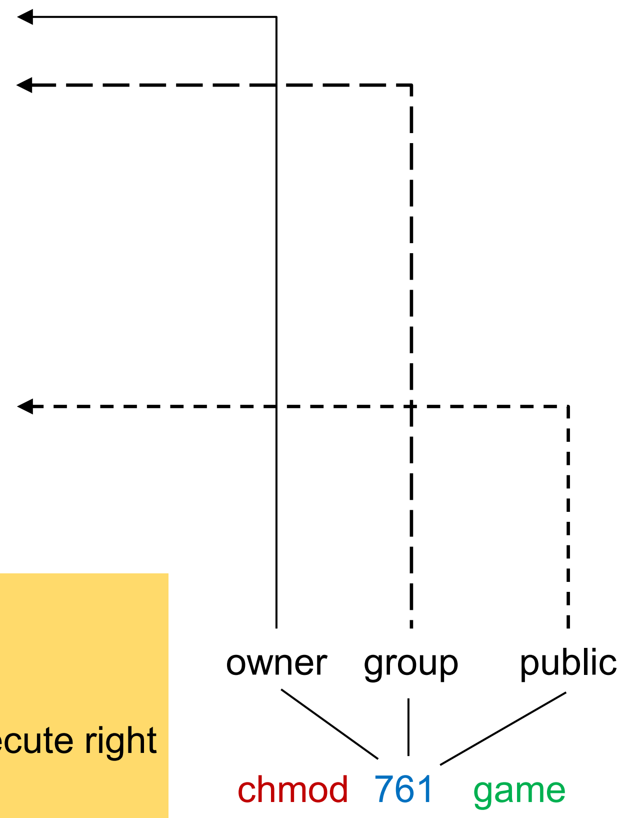
https://www.digitalocean.com/community/tutorials/an-introduction-to-linux-permissions

UNIX access lists and groups

In UNIX, rwx permissions are mapped to a number according to this table

Permission	rwx	sum	number
read, write and execute	rwx	4(r) + 2(w) + 1(x)	7
read and write	rw-	4(r) + 2(w)	6
read and execute	r-x	4(r) + 1(x)	5
read only	r--	4(r)	4
write and execute	-wx	2(w) + 1(x)	3
write only	-w-	2(w)	2
execute only	--x	1(x)	1
none	---	0	0

Example:
Define appropriate access to the file game:
→ owner has full access
→ members of the group developers have read/write but not execute right
→ others (neither owner nor in developers) can only execute

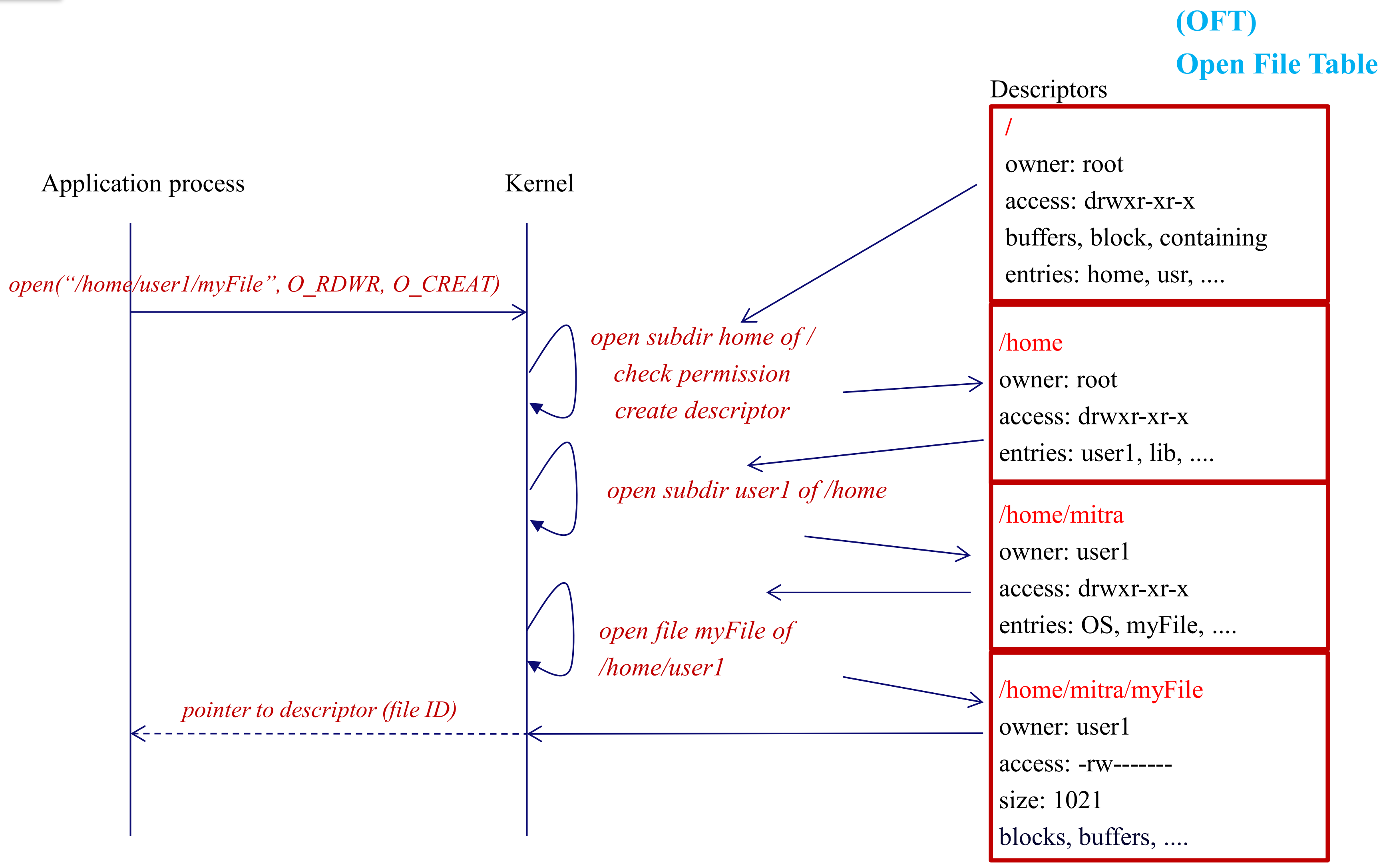


https://en.wikipedia.org/wiki/Chmod

API example: opening a file from an application (unbuffered access)

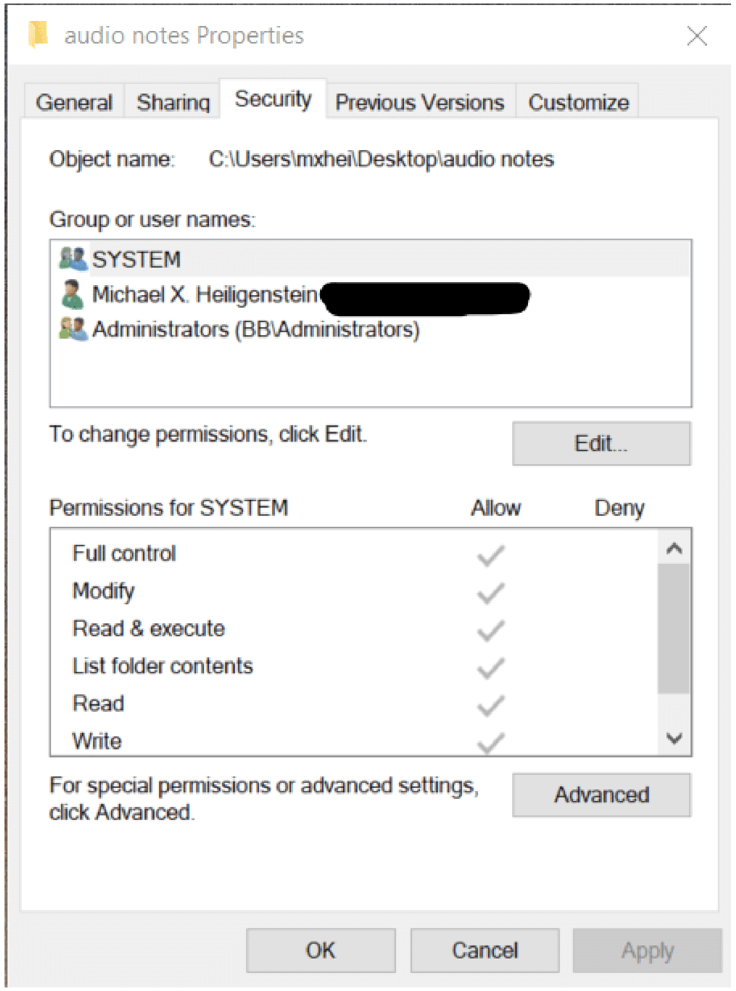
- *fd = open (“*/home/user1/myFile”, *mode, flags)*
  - *mode*: read, write, append, read&write
  - *flags*: steering behavior of *open*, e.g., create file when non-existent
- Recursive open: kernel examines the filename
  - the first directory in ‘/home/user1/myFile’ (i.e., /home) is retrieved
    - passed as an argument to a new call of *open,* on behalf of the kernel
      - or it may be already available as an open file
    - recursively, this open action is performed until the kernel has access to directory /home/user1, including all checks as described
- From the last directory, the kernel finds information about ‘myFile’
  - examines the type, checks whether the caller has **permission**
  - creates a new *descriptor (file handler)* in **Open File Table (OFT)**
    - storing temporary/*dynamic* attributes (e.g., read/write pointers)
  - accesses the file description on the disk using reference obtained from directory
    - caches this into the descriptor
  - (pointer to) descriptor is returned to **user process** for later reference

Conceptual interaction diagram



descriptor = file handler = the pointer used by the user process to manipulate the contents or attributes of the file.

Windows access-control list



audio notes Properties

General | Sharing | Security | Previous Versions | Customize

Object name: C:\Users\mxhei\Desktop\audio notes

Group or user names:

- SYSTEM
- Michael X. Heiligenstein [redacted]
- Administrators (BB\Administrators)

To change permissions, click Edit. Edit...

Permissions for SYSTEM	Allow	Deny
Full control	✓	
Modify	✓	
Read & execute	✓	
List folder contents	✓	
Read	✓	
Write	✓	

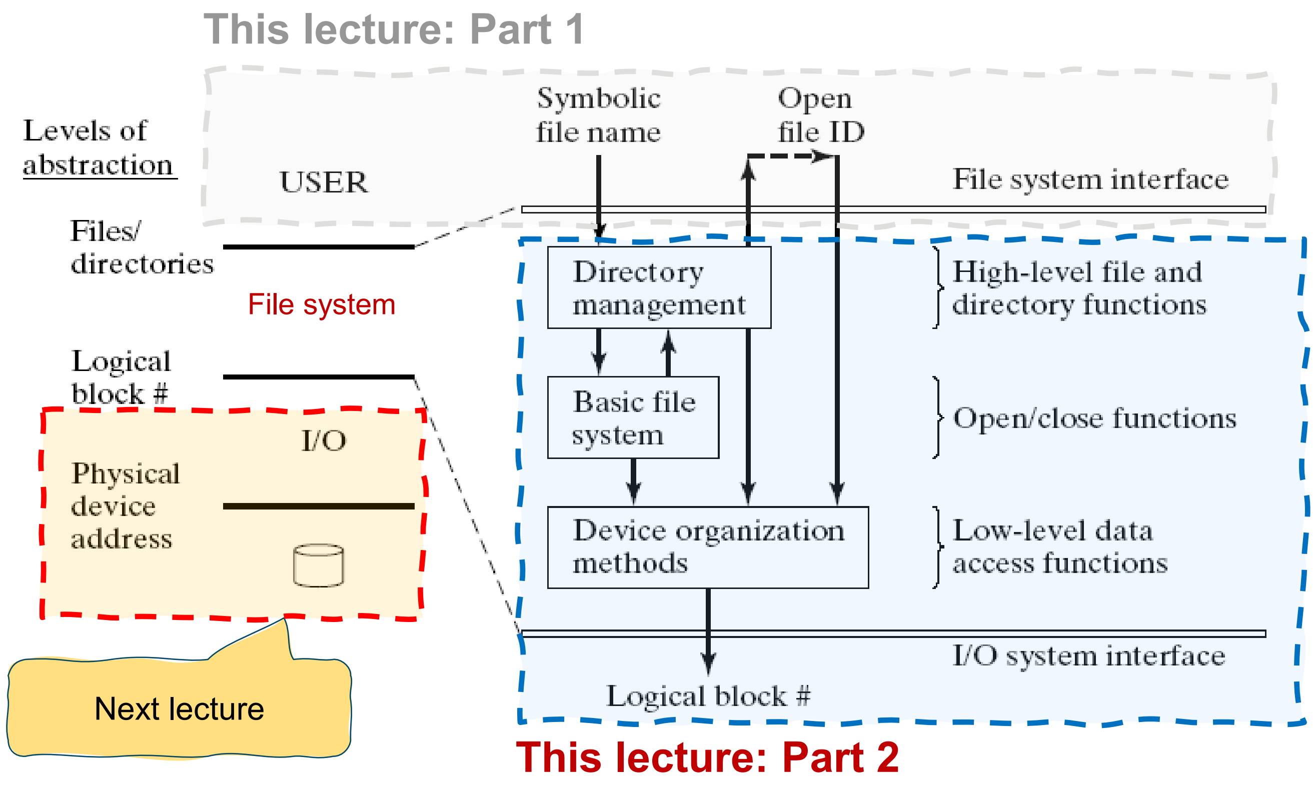
For special permissions or advanced settings, click Advanced. Advanced

OK Cancel Apply

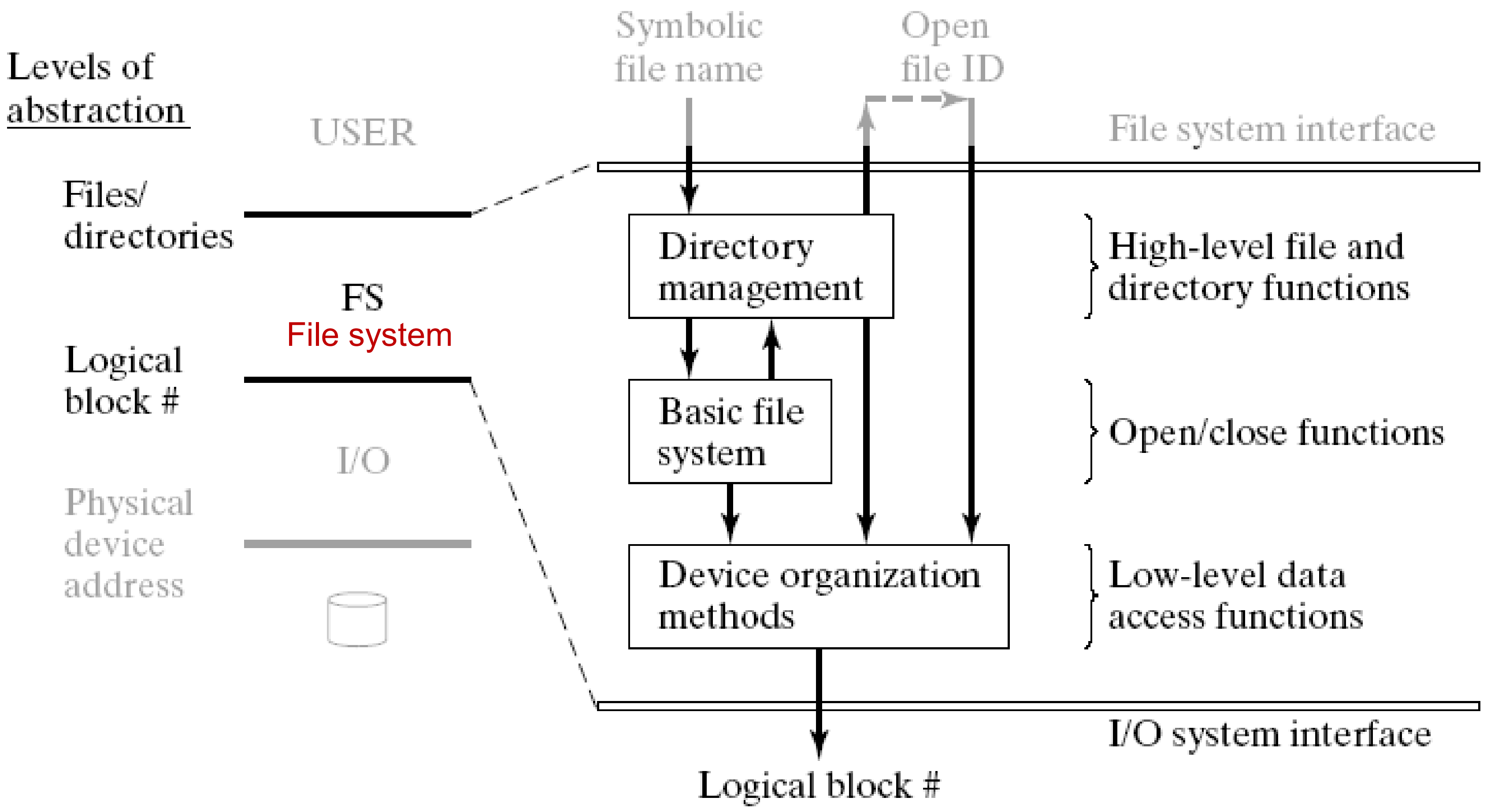
https://firewalltimes.com/access-control-list/

Agenda

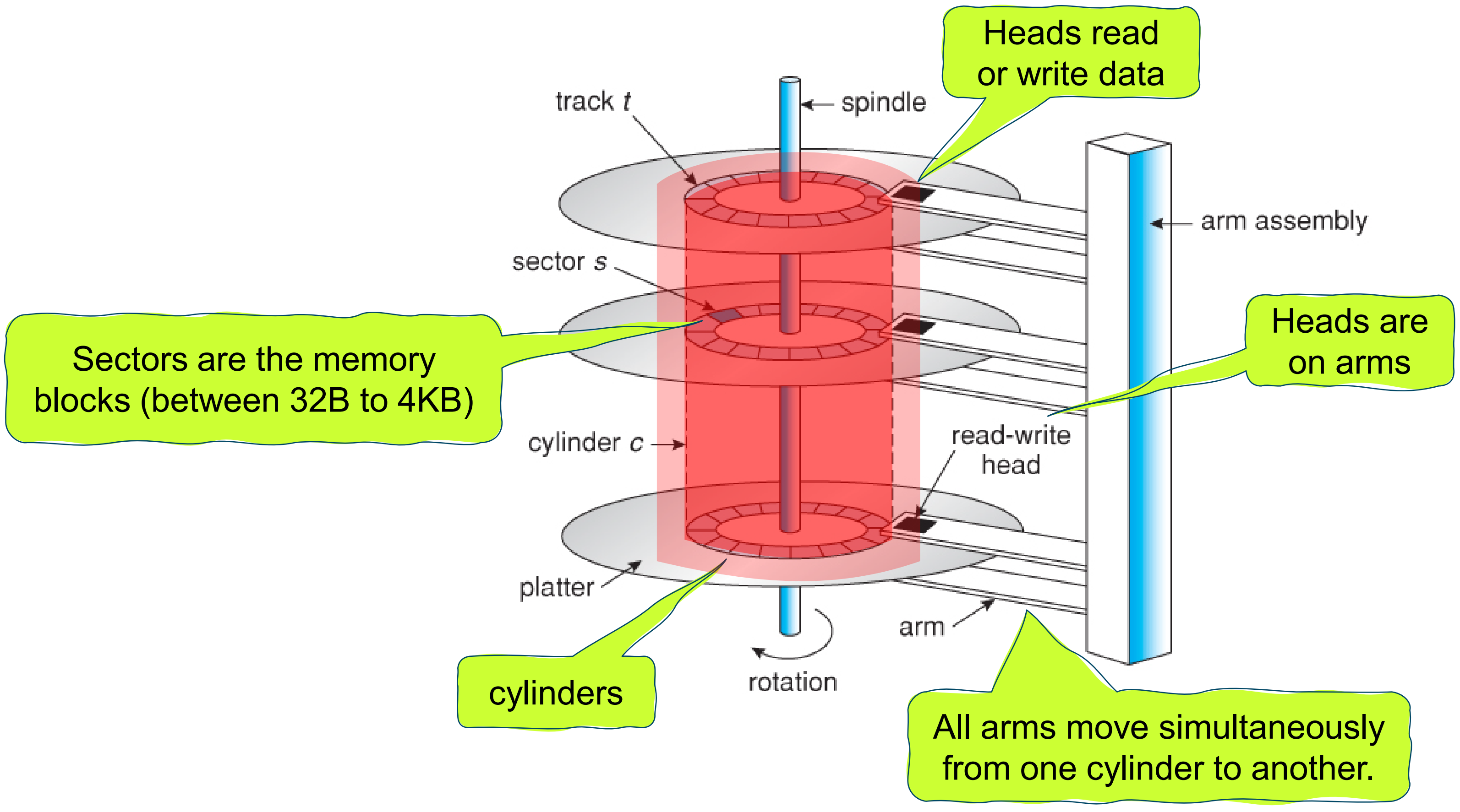
- Objectives of a File System
- Part 1: User view (file system interface)
- **Part 2: OS view (implementation of a file system)**



Layered view



Structure of magnetic disks



Blocks are numbered sequentially from 0 up to the number of sectors on the hard drives.

These are logical blocks.

Booting from the hard drive

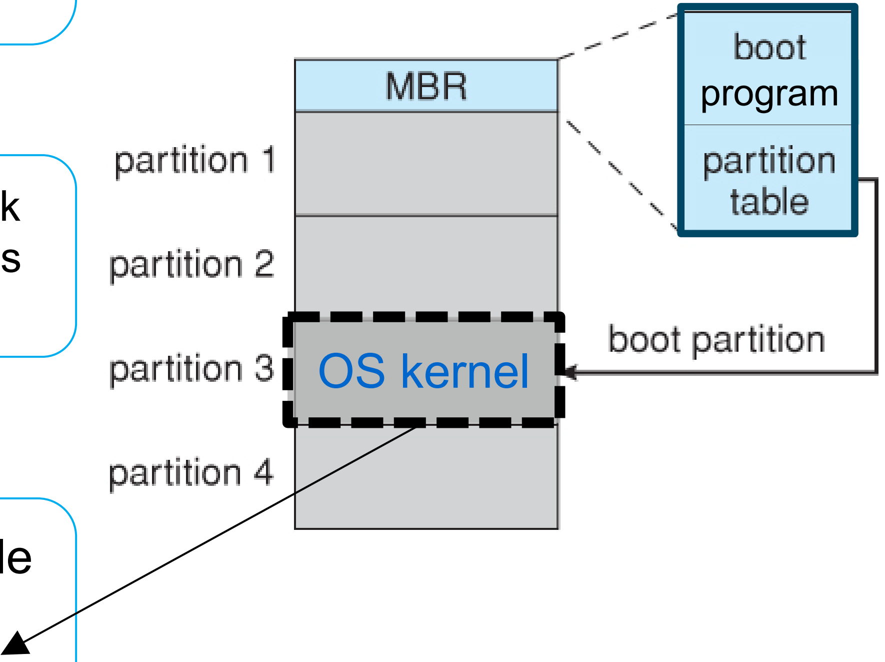
ROM contains a **bootstrap program** that can

1- Find the first sector on the first hard drive (called the **Master Boot Record** or **MBR**), loads it into memory, and then transfer the control to the boot program.

A very small amount of code + the **partition table**

The **partition table** documents how the disk is partitioned into logical disks, and indicates which partition is the boot partition.

2- **The boot program** uses the partition table to find the OS kernel, load it into the main memory, and transfer the control to the **OS**.



Data structures used by the kernel
(stored on the hard disk)

Boot-control block, (per volume) contains information about how to boot the system off of this partition. It is the first sector of the volume.

Volume control block, (per volume) [the **master file table** in UNIX or the **superblock** in Windows] contains information such as the partition table, number of blocks on the filesystem, and pointers to free blocks.

Physical disk	Partition	Filesystem	Drive letter
Hard Disk 1	Partition 1	NTFS	C:
	Partition 2	FAT32	D:
Hard Disk 2	Partition 1	FAT32	E:

In this example,

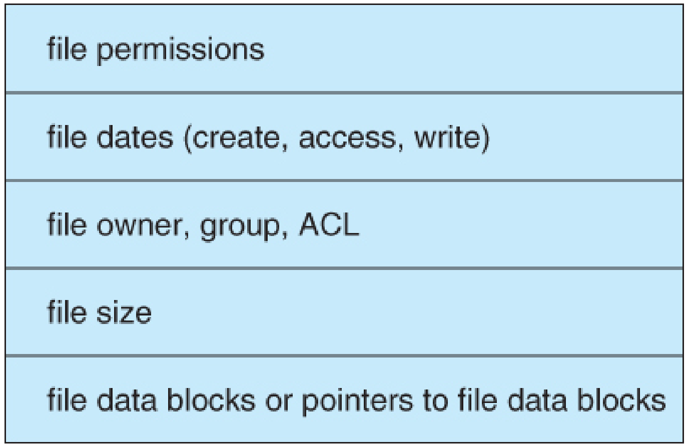
- "C:", "D:", and "E:" are volumes (a.k.a. logical drives).
- Hard Disk 1 and Hard Disk 2 are physical disks.

A volume or logical drive is a storage area with a single file system.

Data structures used by the kernel
(stored on the hard disk)

The kernel maintains a **directory structure** (one per file system), containing file names and pointers to a corresponding **File Control Block (FCB)** for each file.

File Control Block (FCB) (per file) contains details about ownership, size, permissions, …

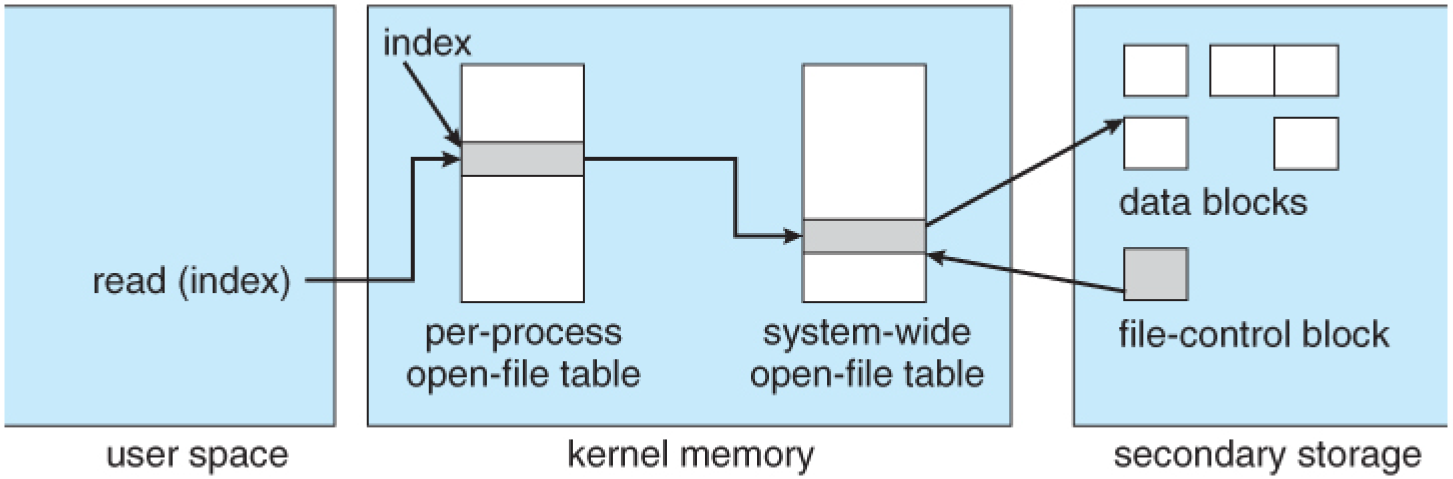

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

- When a new file is created, a new FCB is allocated and filled out with important information regarding the new file.
- The appropriate directory is modified with the new file name and FCB information.

Reading from a file

- When a **process** opens a file, it saves a copy of the FCB of that file from the disk in the system-wide **open file table (OFT).**

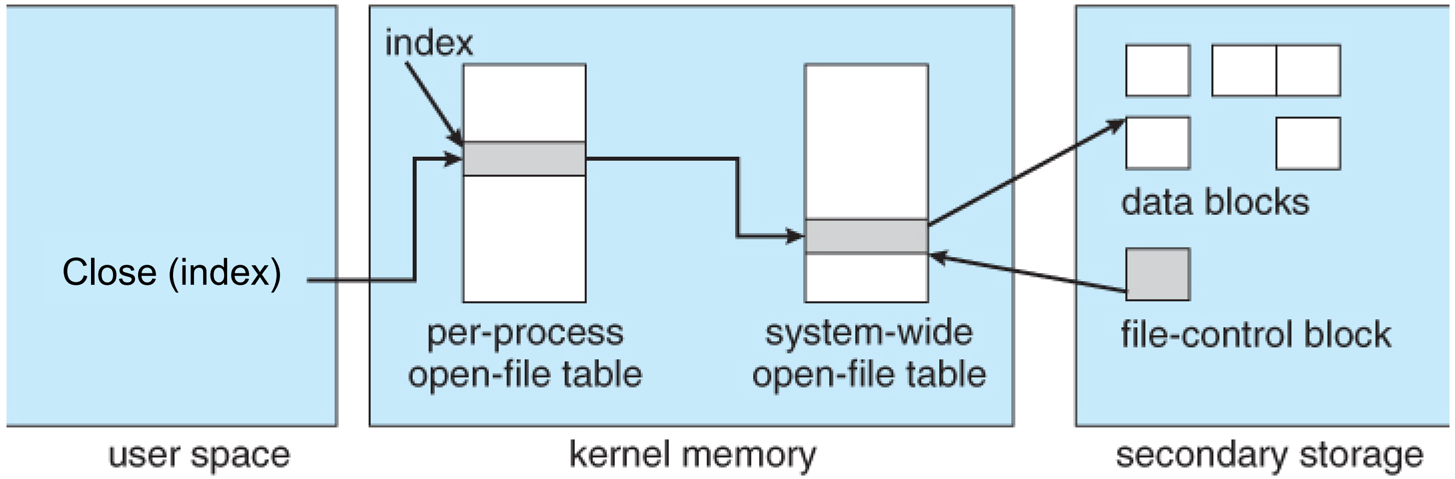
- Additionally, an entry is added to the per-process **open file table** (kept in the PCB) referencing the system-wide **OFT**.



UNIX refers to this index as a file descriptor, and Windows refers to it as a file handle.

Opening an already open file and closing files

- If several processes open the same file, there will be only one entry for that file in the system-wide OFT, and one entry in the per-process OFT of each of those processes.
  - System-wide OFT has a counter keeping track of how many processes opened that file.
- When a file is **closed** in a process, the per-process OFT entry is freed, and the counter in the system-wide OFT is decremented.
- If the counter reaches zero, no process has the file open so the FCB can be removed from the system-wide OFT.



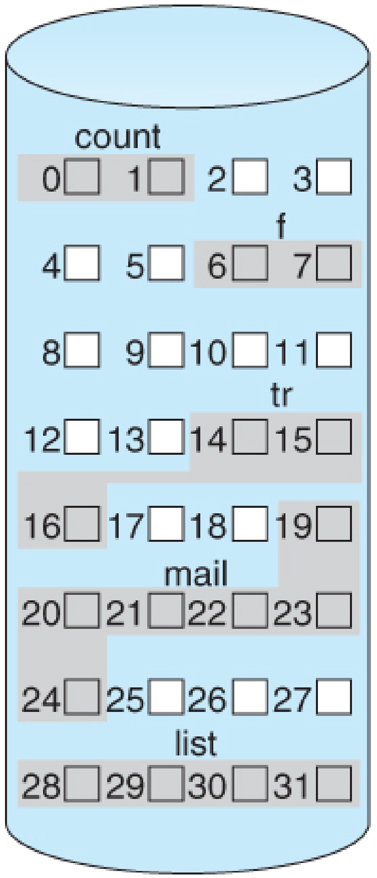
Storing files on the disk

- FCB keeps track of where a file is saved on the secondary memory.
- There are several ways a file can be stored on the secondary memory

“Contiguous”
allocation of blocks per file

“Linked”
allocation of blocks per file

“Indexed”
allocation of blocks per file

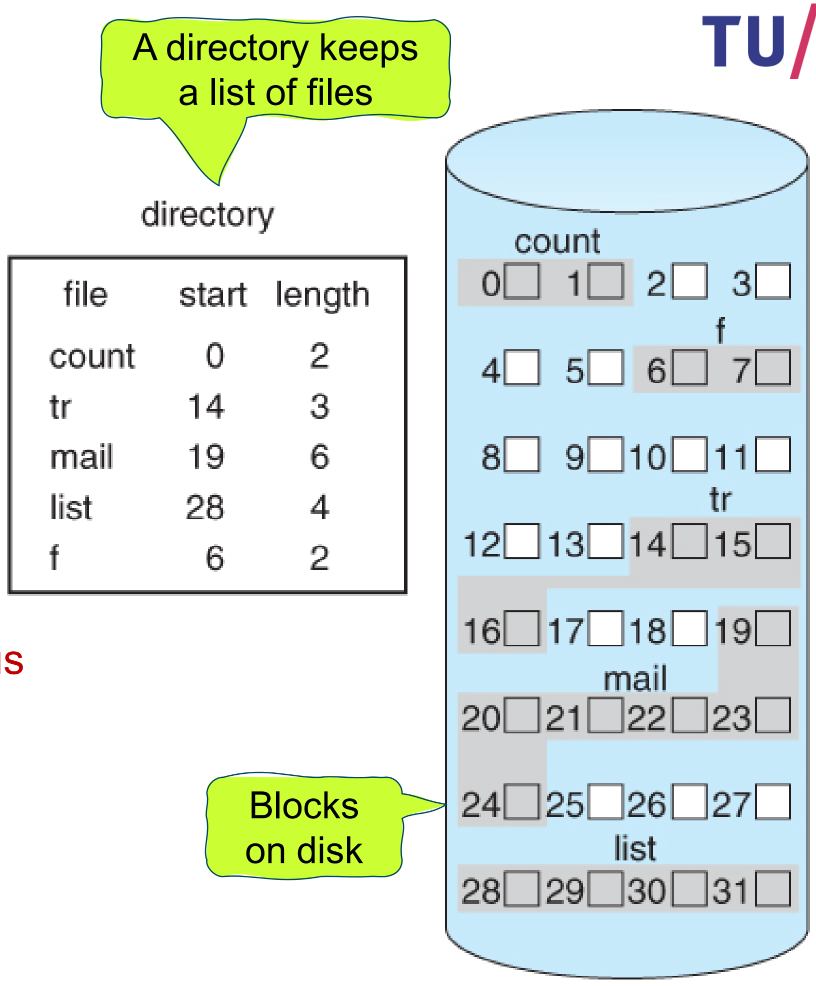


Storing files on disk:
"Contiguous" allocation of blocks

all blocks of a file are kept together contiguously on the disk.

- Very fast when reading successive blocks of the same file.

- Adding a new file requires finding a contiguous space that is big enough for the file.
- **Strategies**: best fit, first fit, next fit,…



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

How to deal with the growth of a file?

1. **Relocate:** A lot of overhead.
2. **Over-approximate the space during an allocation (instead of allocating X bytes, allocate X + Y bytes. It allows file to grow by Y bytes without a need for relocation):**
   - Downside: wasting disk space if files do not grow. Also, unclear how to approximate.
3. **Allocate large contiguous chunks called "extent" (when one is filled, allocate a new extent).**
   - Adopted by file systems used in high-performance computing.

Storing files on disk:
"Contiguous" allocation of blocks

Example: First-fit policy

- Assume that the last file added was "mail".
- Assume that the file system uses **first-fit strategy** to find just the right amount of space for a new file.

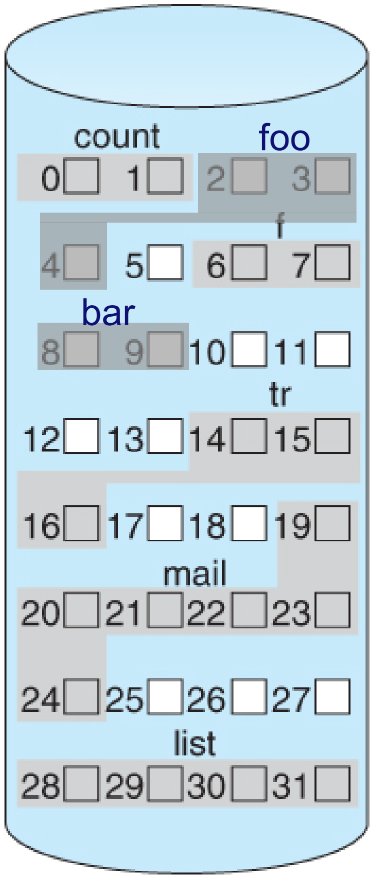
- **Which blocks will be allocated to the file "foo" of size 3?**
- **After adding "foo", which blocks will be allocated to the file "bar" of size 2?**
- **After adding "foo" and "bar", which blocks will be allocated to the file "zee" of size 6?**

zee cannot be added to this cylinder unless a defragmentation operation puts together all the free space.

First-fit policy tries to find the "first" space that is large enough for the file. It searches from block 1 to the last block.

directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2
foo	2	3
bar	8	2



Storing files on disk:
"Contiguous" allocation of blocks

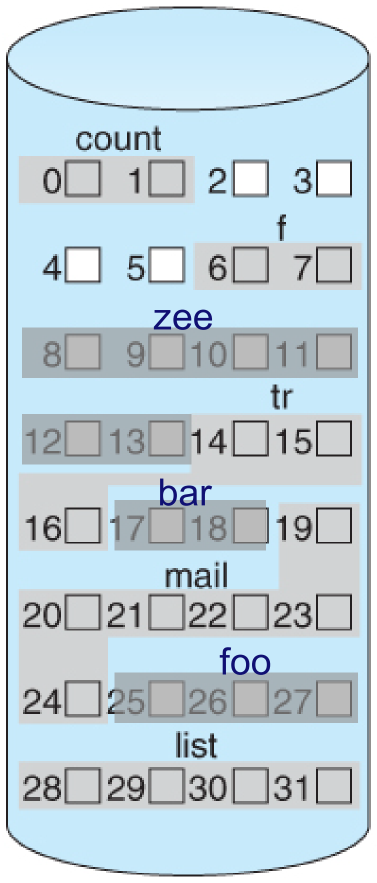
Example: Best-fit policy

- Assume that the last file added was "mail".
- Assume that the file system uses **best-fit strategy** to find just the right amount of space for a new file.

- **Which blocks will be allocated to the file "foo" of size 3?**
- **After adding "foo", which blocks will be allocated to the file "bar" of size 2?**
- **After adding "foo" and "bar", which blocks will be allocated to the file "zee" of size 6?**

directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2
foo	25	3
bar	17	2
zee	8	6



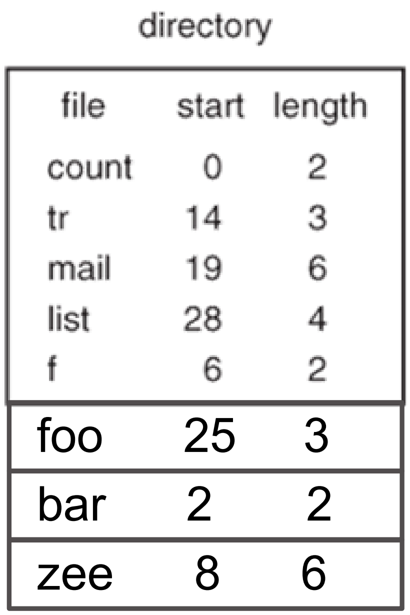
Best-fit policy tries to find the smallest available space that is large enough to accommodate the file, thereby minimizing wasted space. It searches from block 1 to the last block.

Storing files on disk:
“Contiguous” allocation of blocks

Example: Next-fit policy

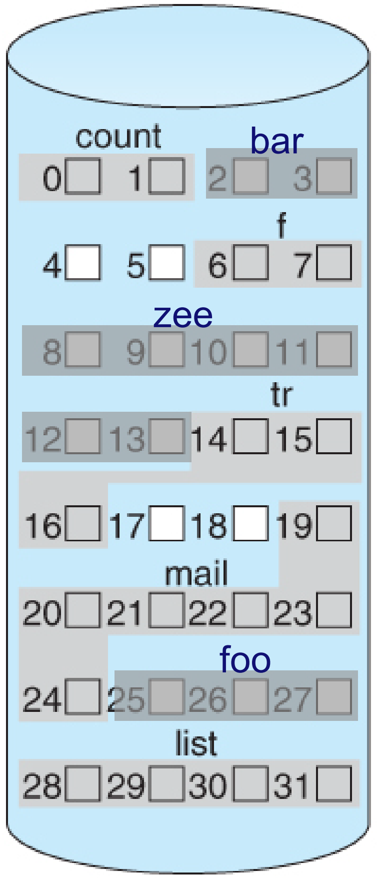
- Assume that the last file added was “mail”.
- Assume that the file system uses **next-fit strategy** to find just the right amount of space for a new file.

- **Which blocks will be allocated to the file “foo” of size 3?**
- **After adding “foo”, which blocks will be allocated to the file “bar” of size 2?**
- **After adding “foo” and “bar”, which blocks will be allocated to the file “zee” of size 6?**



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2
foo	25	3
bar	2	2
zee	8	6



Next-fit policy tries to find the “next” available space that is large enough to accommodate the file. It “remembers” the location of the last added file and starts its search from there. If it reaches the end, it starts from the beginning.

Storing files on the disk

“Contiguous”
allocation of blocks per file

“Linked”
allocation of blocks per file

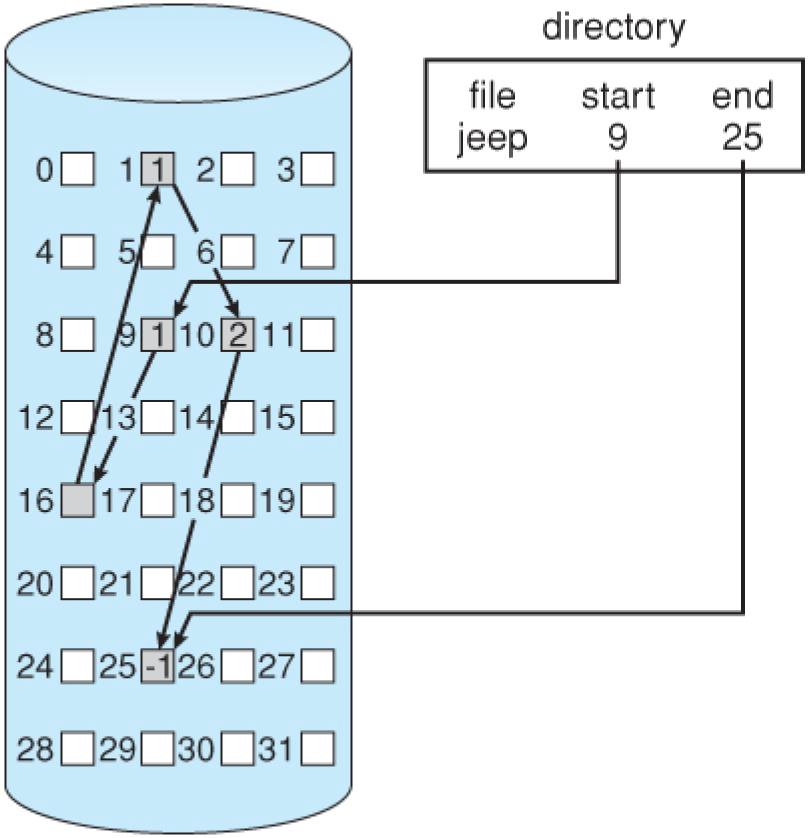
“Indexed”
allocation of blocks per file

Storing files on disk:
“Linked” allocation of blocks

- Each block has a pointer to the next block.
- The directory keeps track of the starting and ending blocks.

- No need to know the file size in advance
- Allows files to grow dynamically at any time

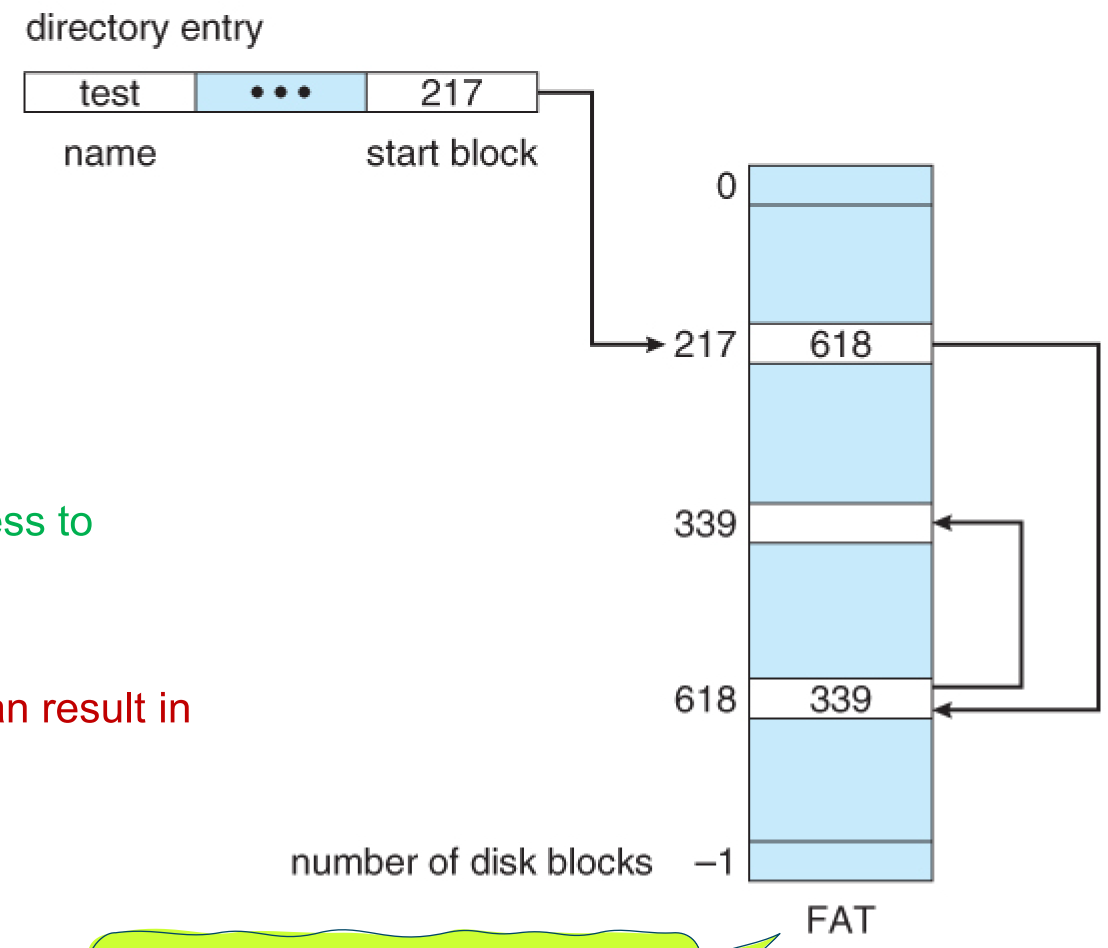
1. Only efficient for sequential access.
2. Random access requires starting at the beginning of the list **for each new location access!**
3. If a pointer is lost or damaged, the rest of the file is unrecoverable.



Storing files on disk:
"Linked" allocation of blocks

FAT (file allocation table) used by DOS and Windows:

- The links are stored in a **separate table** (called FAT) at the beginning of the disk.



Size of the FAT table depends on the number of blocks (each entity contains a block number).

- The FAT table can be cached in memory, therefore random access to the files can be much faster.

- Any damage to the FAT table can result in the loss of file blocks.

Storing files on the disk

“Contiguous”
allocation of blocks per file

“Linked”
allocation of blocks per file

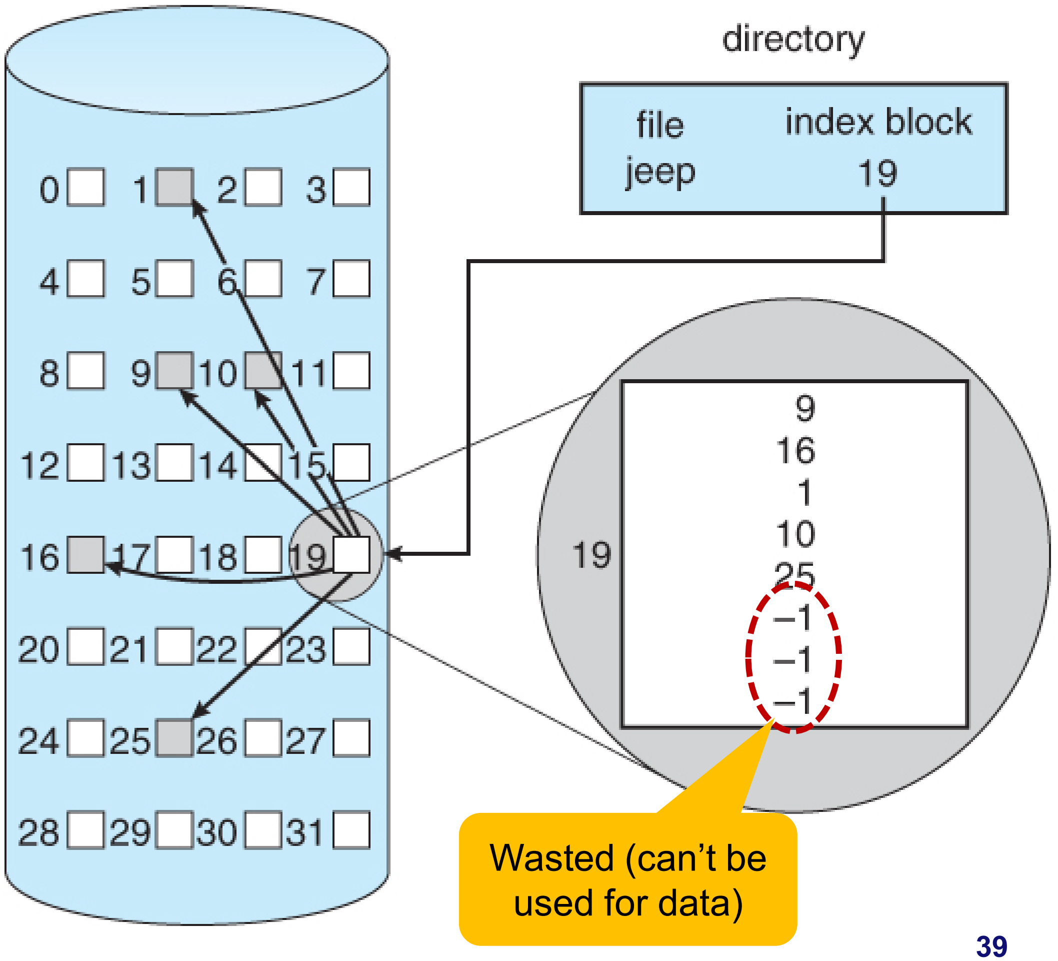
“Indexed”
allocation of blocks per file

Storing files on disk:
"Indexed" allocation of blocks

Combines **all of the indexes for accessing each file into a common block** (for that file), as opposed to spreading them all over the disk or storing them in a FAT table.

- Fast random access
- Easily handles dynamic files (that grow or shrink during their life-times)
- No need to know the file size in advance.
- No need for a separate table (outside the memory space of the file) to keep track of all data blocks.

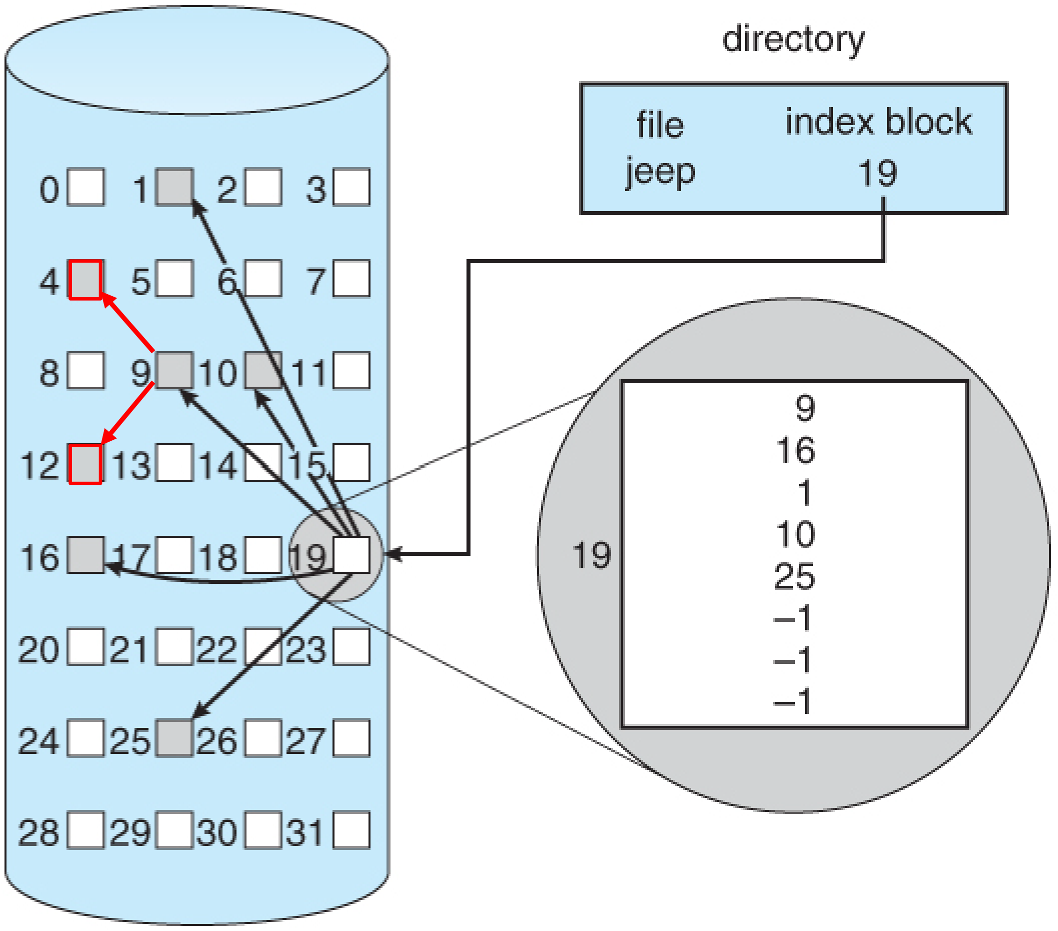
- Some disk space is wasted because an entire **index block** must be allocated for each file, regardless of how many **data blocks** the file contains.



Storing files on disk:
"Indexed" allocation of blocks

A single indexed block limits the maximum file size that can be stored. Therefore, some operating systems such as UNIX use **multi-level indexing**.

The **first index block** contains a set of pointers to **secondary index blocks**, which in turn contain pointers to the actual data blocks.



Storing files on disk:
"Indexed" allocation of blocks

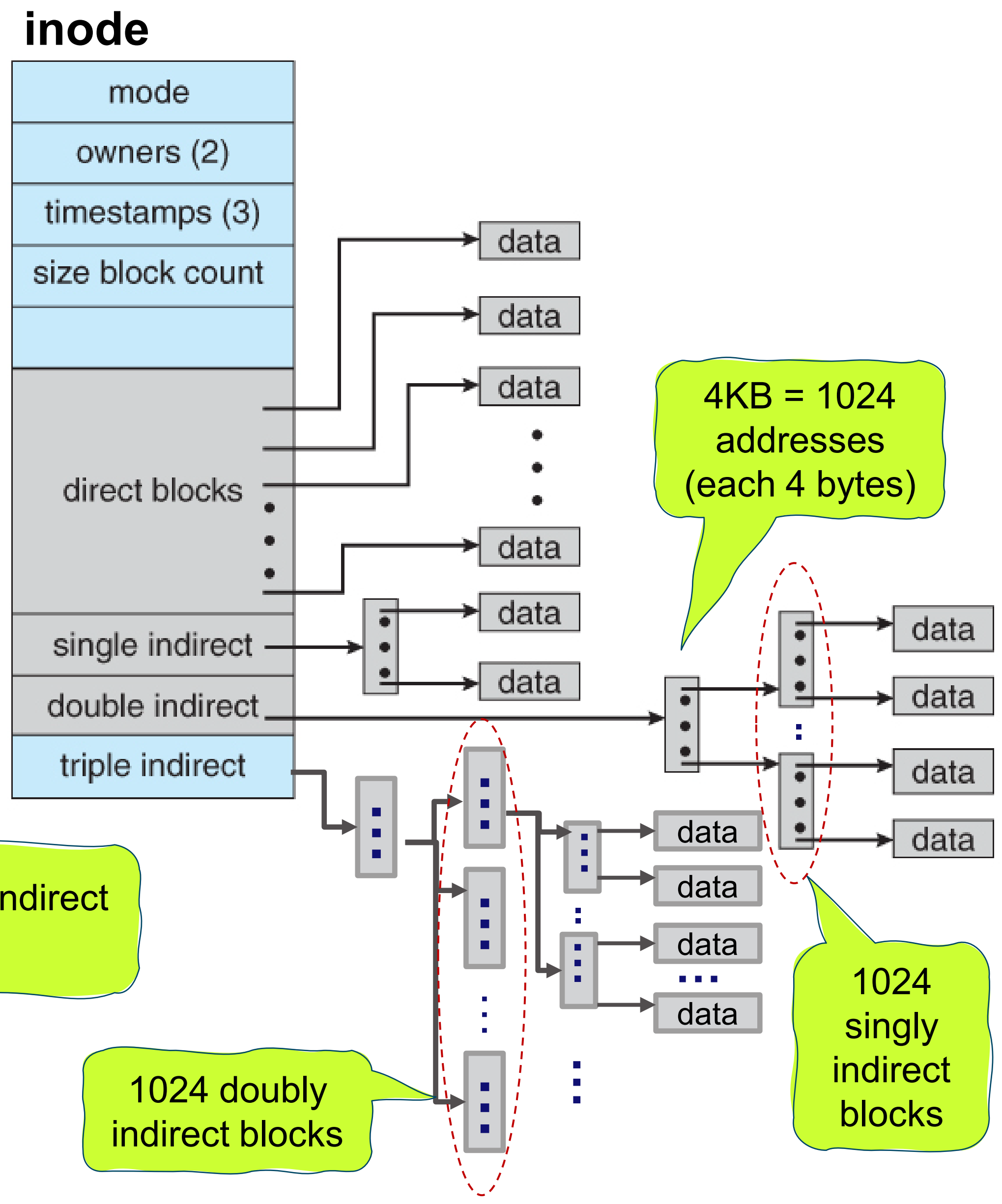
UNIX (inode) and EXT4 file system:

- The first 12 **data block pointers** are stored directly in the inode.
- Then there are Singly, Doubly, and Triply **indirect pointers**, each providing access to a single, double, and triple levels of pointers.

+ Very fast for small files with less than 12 data blocks (i.e., files smaller than 48KB -- with 4KB block sizes)

- Fast for files up to 4144KB (they only need a single indirect block which can be cached).

4144K = 12x4K (direct blocks) + 1024x4K (indirect blocks via 1 single indirect block)



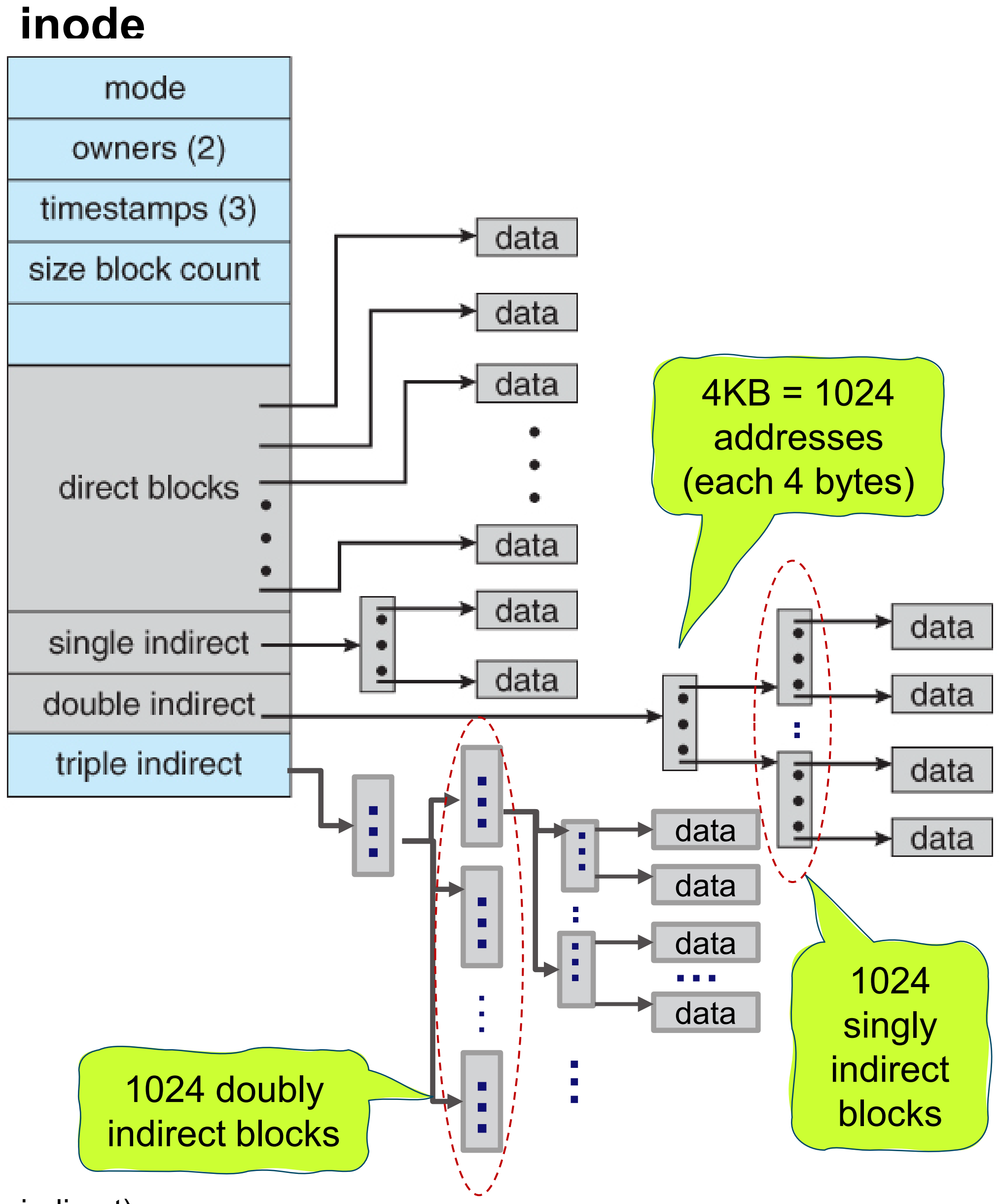
Storing files on disk:
"Indexed" allocation of blocks

UNIX (inode) and EXT4 file system:

- The first 12 **data block pointers** are stored directly in the inode.
- Then there are Singly, Doubly, and Triply **indirect pointers**, each providing access to a single, double, and triple levels of pointers.

+ Very fast for small files with less than 12 data blocks (i.e., files smaller than 48KB -- with 4KB block sizes)

- Fast for files up to 4144KB (they only need a single indirect block which can be cached).
- Huge files are still accessible using a relatively small number of disk accesses (larger in theory than can be addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers.)

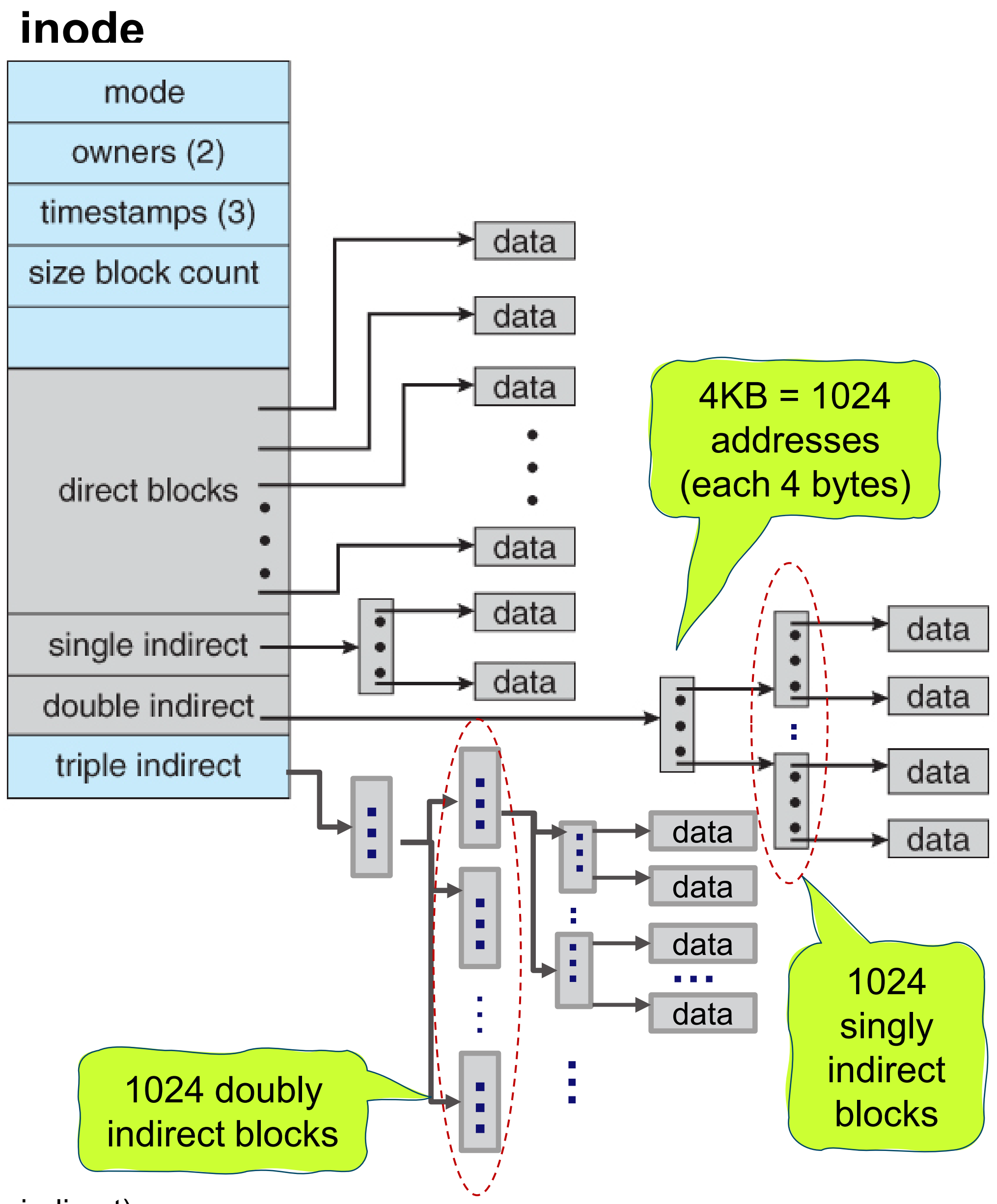


Note: 4144KB = 12*4K (direct) + 1024*4KB (singly- indirect)

Example: inode

How would an inode keep track of a file with 10000KB?

(Assume that blocks are 4KB and addresses are 4 bytes each.)

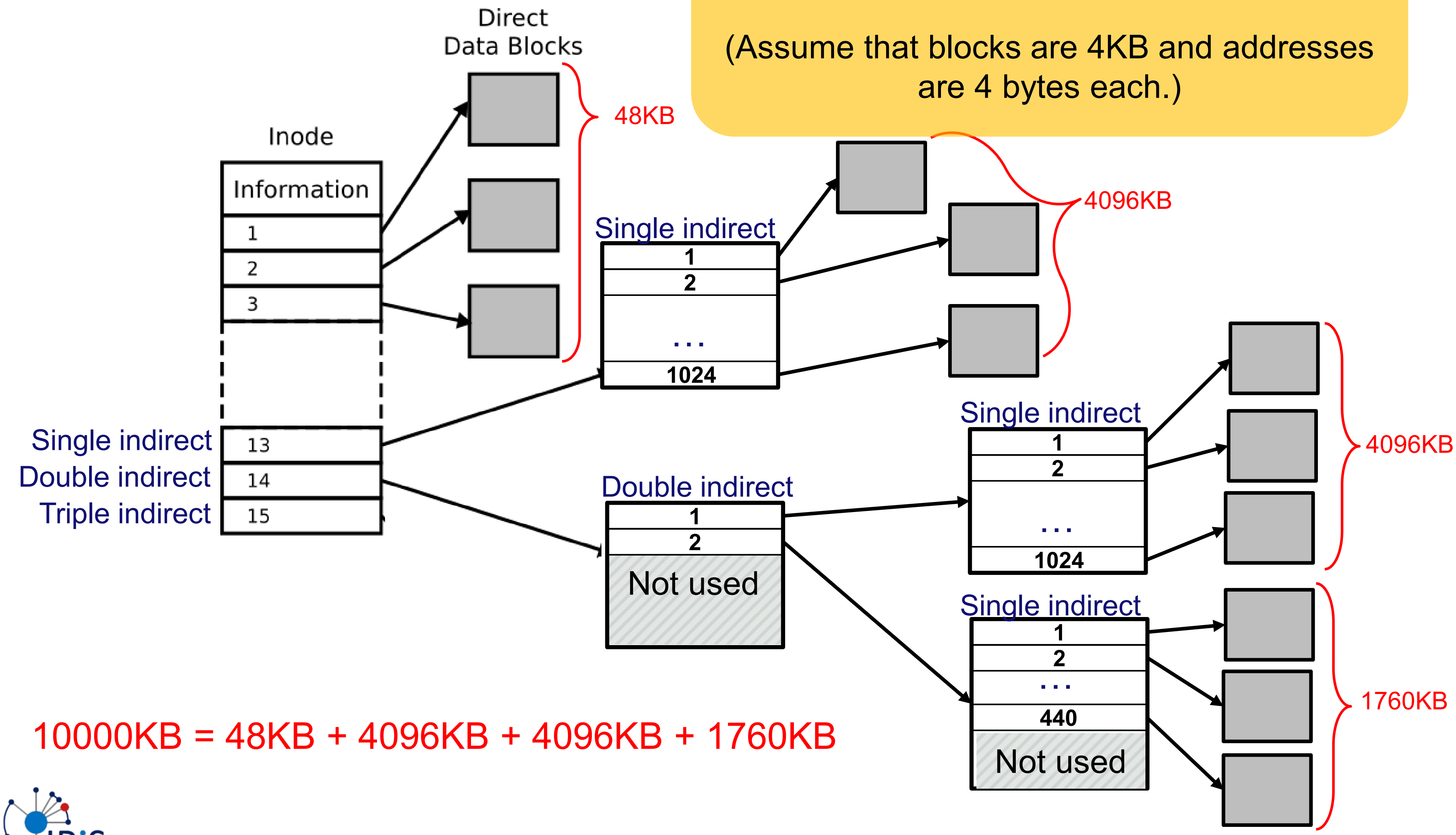


Note: 4144KB = 12*4K (direct) + 1024*4KB (singly- indirect)

UNIX inode: an example

How would an inode keep track of a file with 10000KB?

(Assume that blocks are 4KB and addresses are 4 bytes each.)



10000KB = 48KB + 4096KB + 4096KB + 1760KB

UNIX inode: an example

How would an inode keep track of a file with 10000KB?

(Assume that blocks are 4KB and addresses are 4 bytes each.)

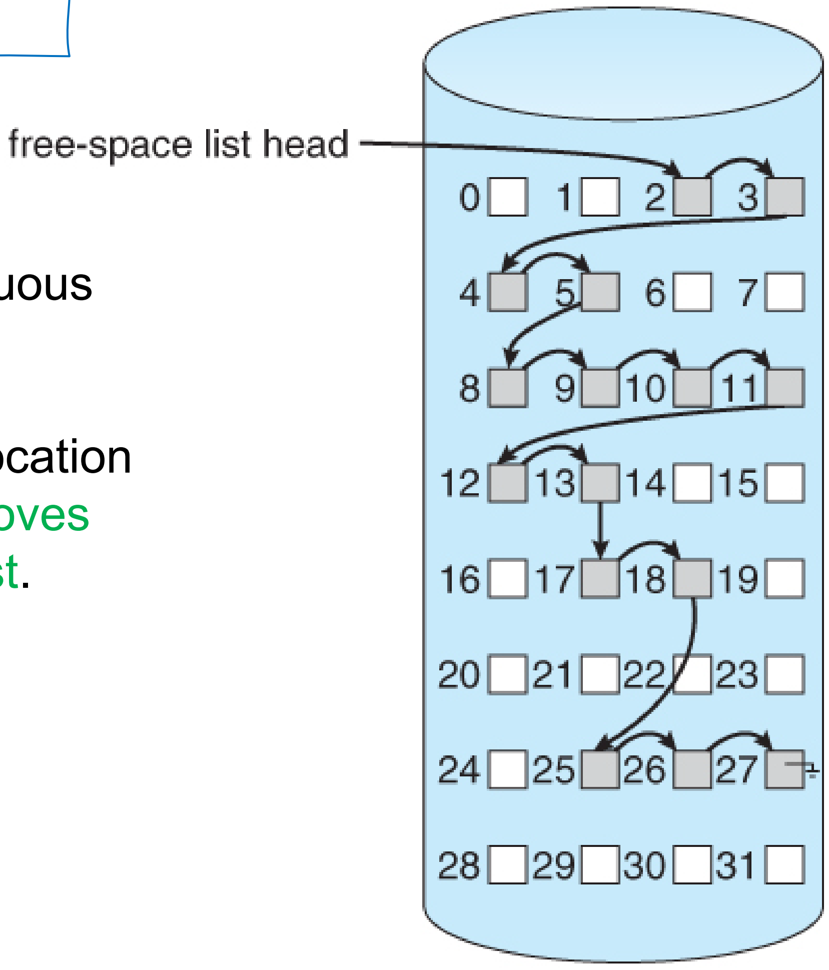
- With a 4KB block size, we can store 1024 addresses (each 4 bytes).
- The 12 direct blocks will be used for the first 48KB of the file. For the remaining 10000-48 = 9952KB, the inode will use the singly and doubly address blocks (starting with singly).

- A singly address block contains addresses of 1024 data blocks. Each data block stores 4KB of data of the file.
- Therefore, 1024 * 4KB = 4096KB of data will be maintained using the singly address block.
- The remaining 10000 – 48 - 4096KB = 5856KB of data should be stored using doubly indirect blocks.
- One doubly indirect block contains the address of 1024 singly indirect blocks each of which can be used to store 4096KB.
- To store 5856KB, we will need more than one doubly indirect block.

Free space management

Free space is kept track of by a linked list.

- Traversing the list and/or finding a contiguous block of a given size are not easy.
- Under the "**linked**" and "**indexed**" file allocation strategies, the system just adds and removes single blocks from the beginning of the list.



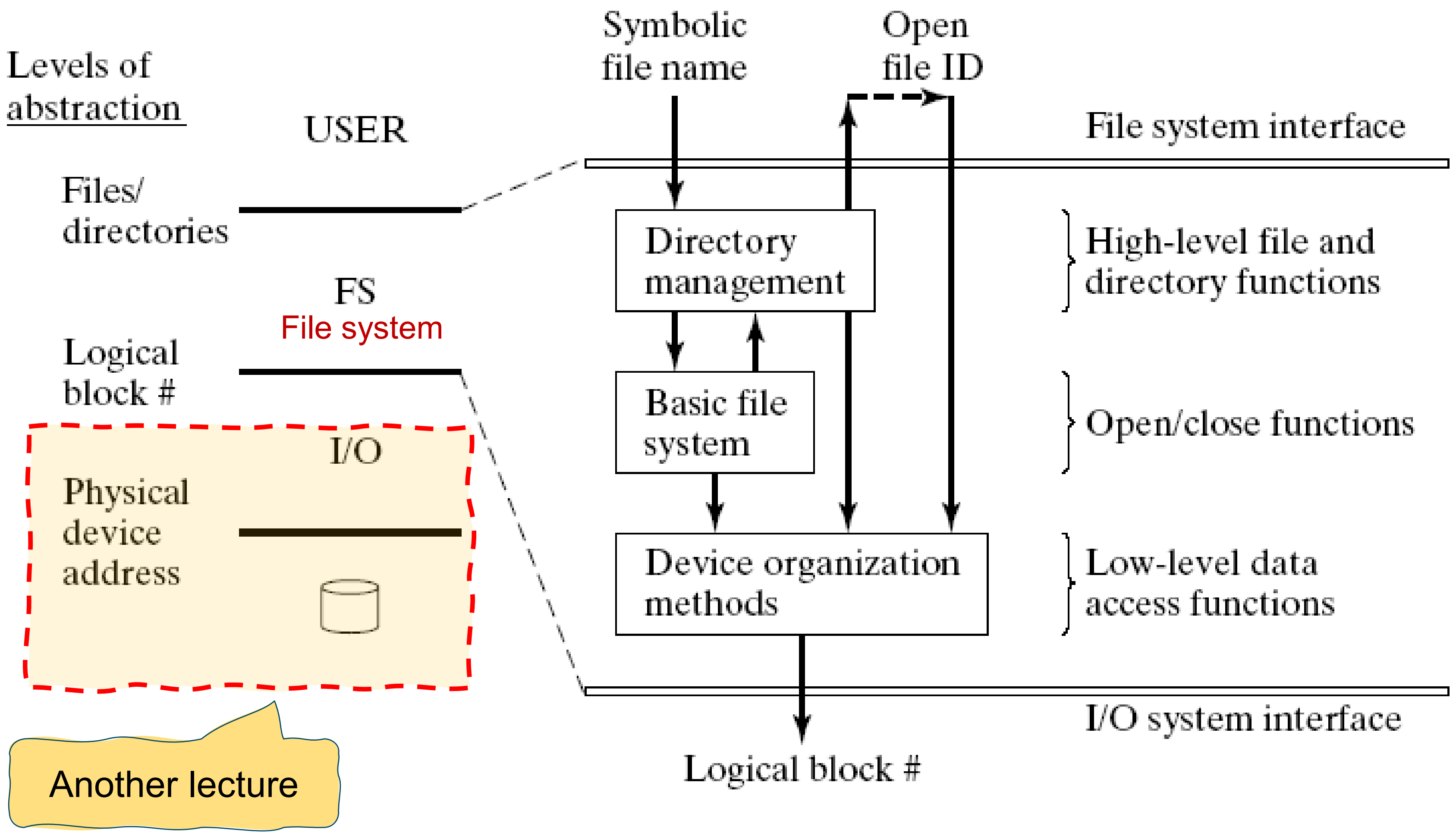
Example: File system implementations from Microsoft

- **File Allocation Table (FAT32)**
  - max file size: 4GB
    - not good for storing larger files
  - max volume size: 2TB
    - larger hard drives must be formatted into 2TB FAT32 partitions
- Block allocation strategy: **linked**

- **New Technology File System (NTFS)** - improvements over FAT32
  - support for huge files (up to 2^{64} bytes in theory)
  - support for huge volumes (~2^{64} clusters & max 64KB per cluster)
  - journaling for recording log of updates
  - increased security (file permissions)
  - Uses a Master File Table to keep track of blocks
  - Uses “extents” to keep file’s data in sequential blocks

We already saw UNIX file system (UFS) which uses the inodes.

Summary


Levels of abstraction
USER
Files/ directories
FS
File system
Logical block #
I/O
Physical device address
Another lecture
Symbolic file name
Open file ID
File system interface
Directory management
Basic file system
Device organization methods
High-level file and directory functions
Open/close functions
Low-level data access functions
I/O system interface
Logical block #